

Part 1 – Example 1

First Steps of any analysis

- Snapshot your vm before beginning any analysis
- Hash all files that you plan to analyze
- Keep a log of steps taken

Opening Comments

- Easy to determine maliciousness of malware
- But to determine the purpose we must be willing to analyze on a more granular level
- This allows better IOC's and stronger understanding of the purpose and behavior

Analysis Routine

1) Static Analysis

- Strings
- Imports
- Exports

2) Dynamic Analysis

- Set breakpoints
- Analyze data, memory, and functions at runtime

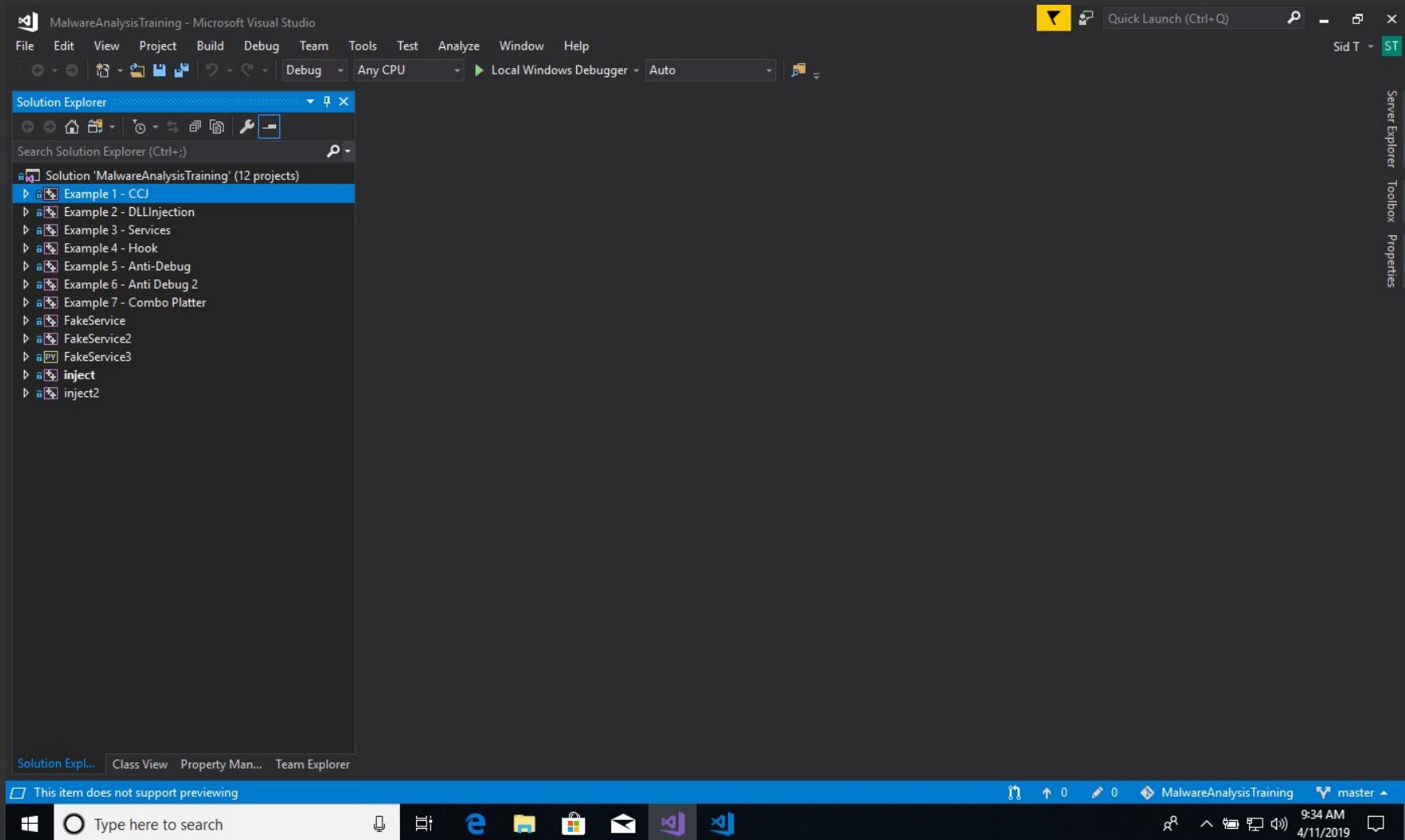
3) Update your timeline.

4) Back to Step 1.

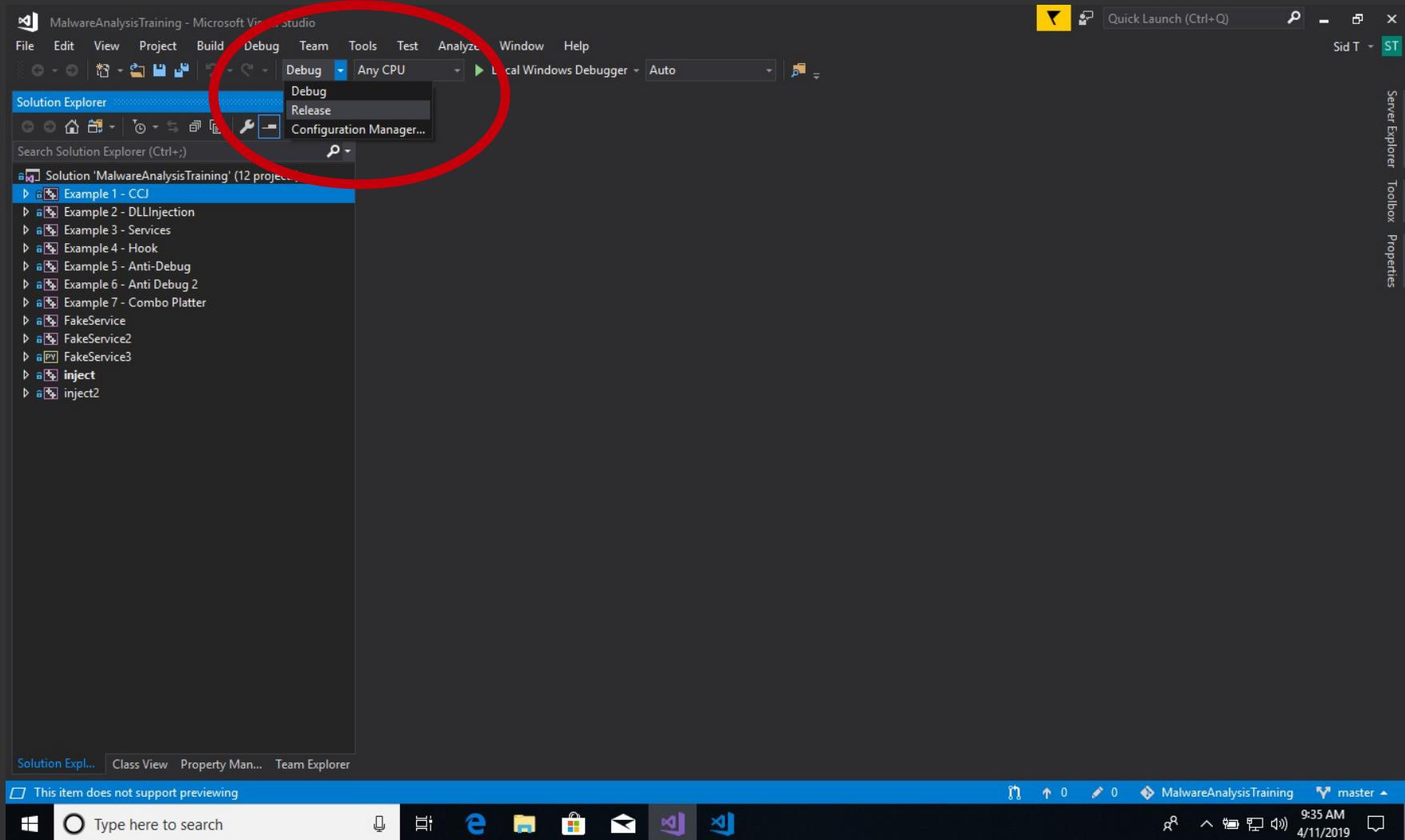
Setting up the First Example

(The Windows VM will be required at this point with the project opened and ready in Visual Studio. Read the Requirements.pdf for setting up the project.)

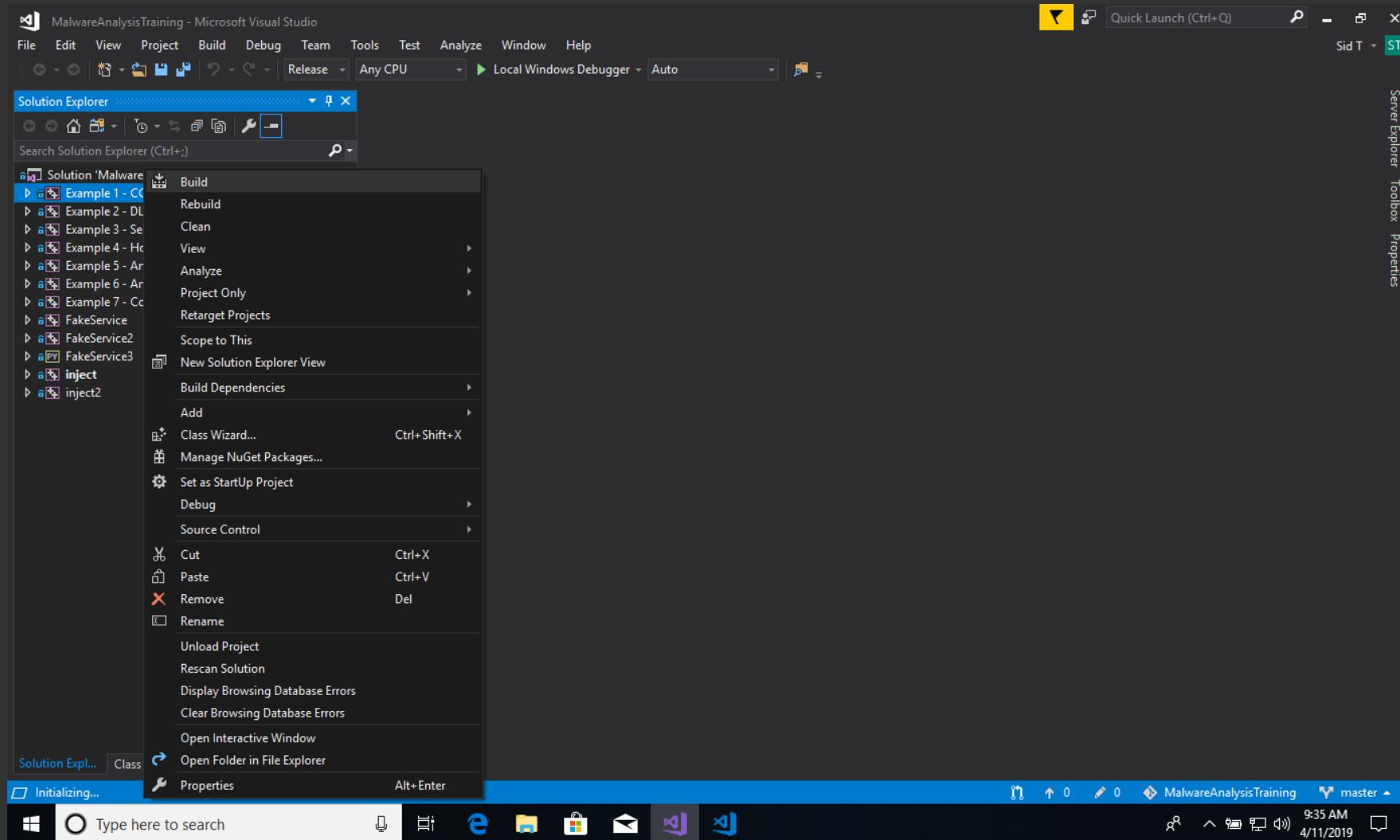
This is what the Visual Studio project should look like if opened up properly.



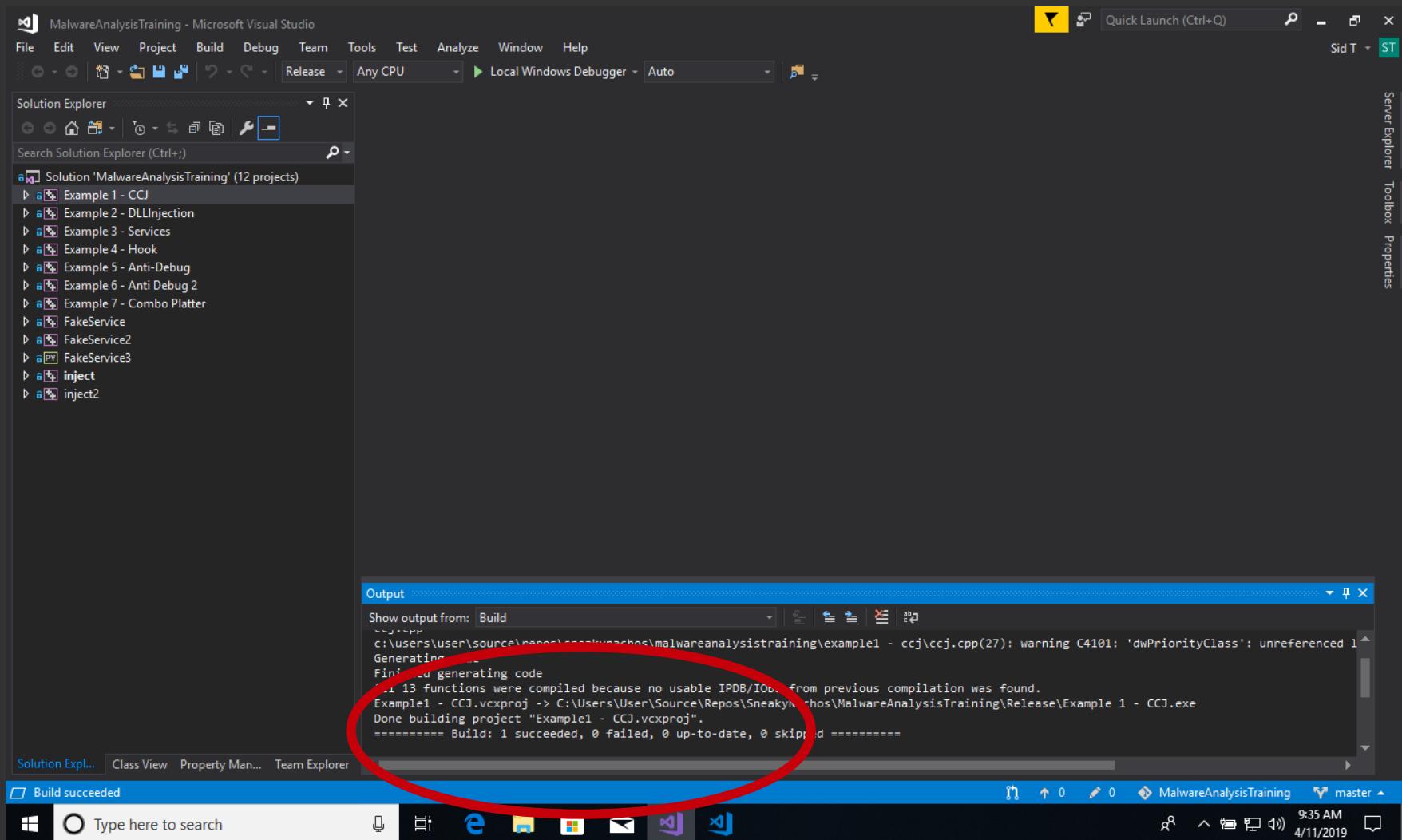
We first need to set the build to Release. Shown below is where the drop down is to switch between Debug/Release Builds. “Any CPU” should also be set to the target architecture of the system (x86) if 32-bit and (x64) if 64-bit.



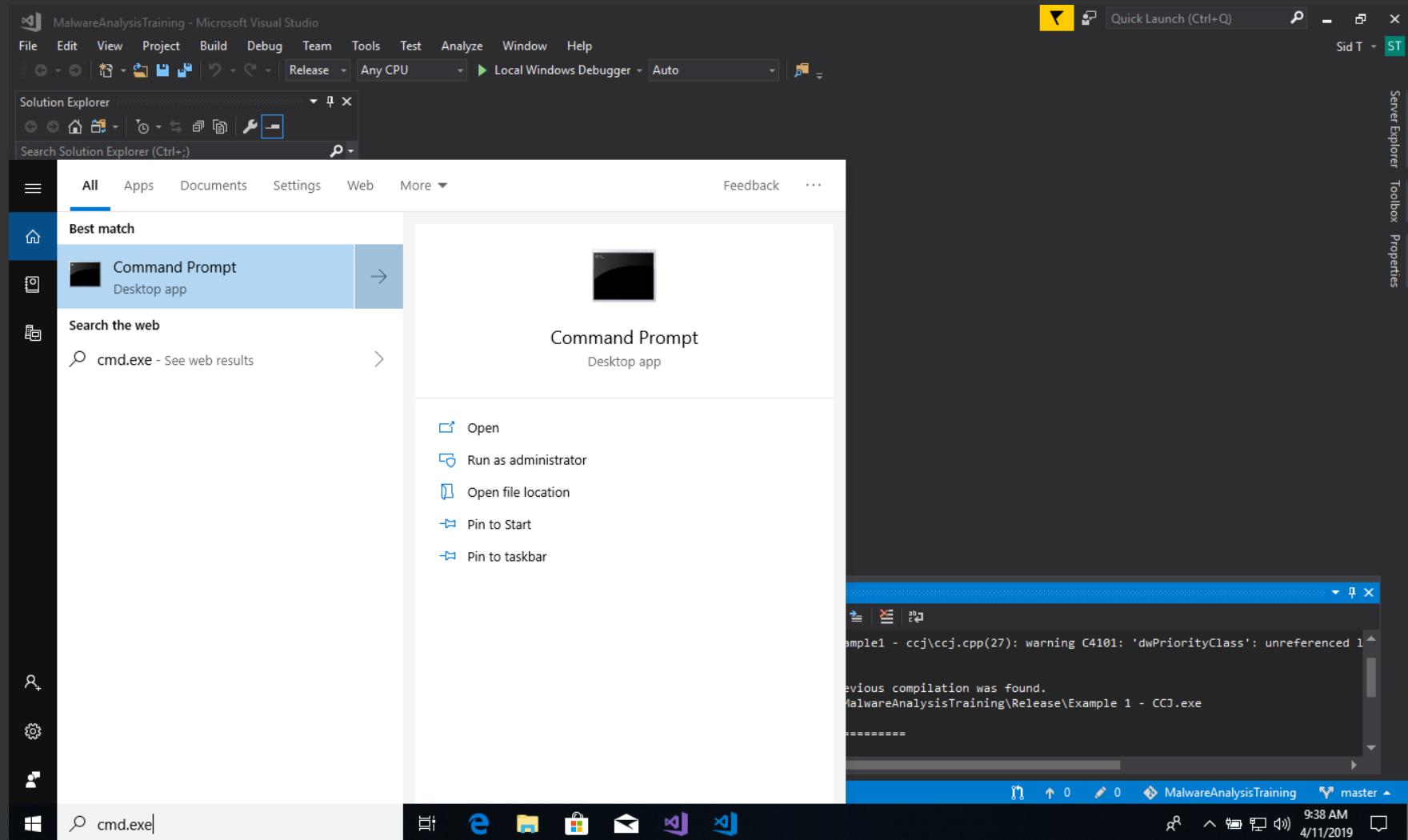
Hover over “Example 1 – CCJ” and right click to open up the various drop down options for the project. At the top there will be a “Build” option. Go ahead and click build and wait for the compiler to startup.



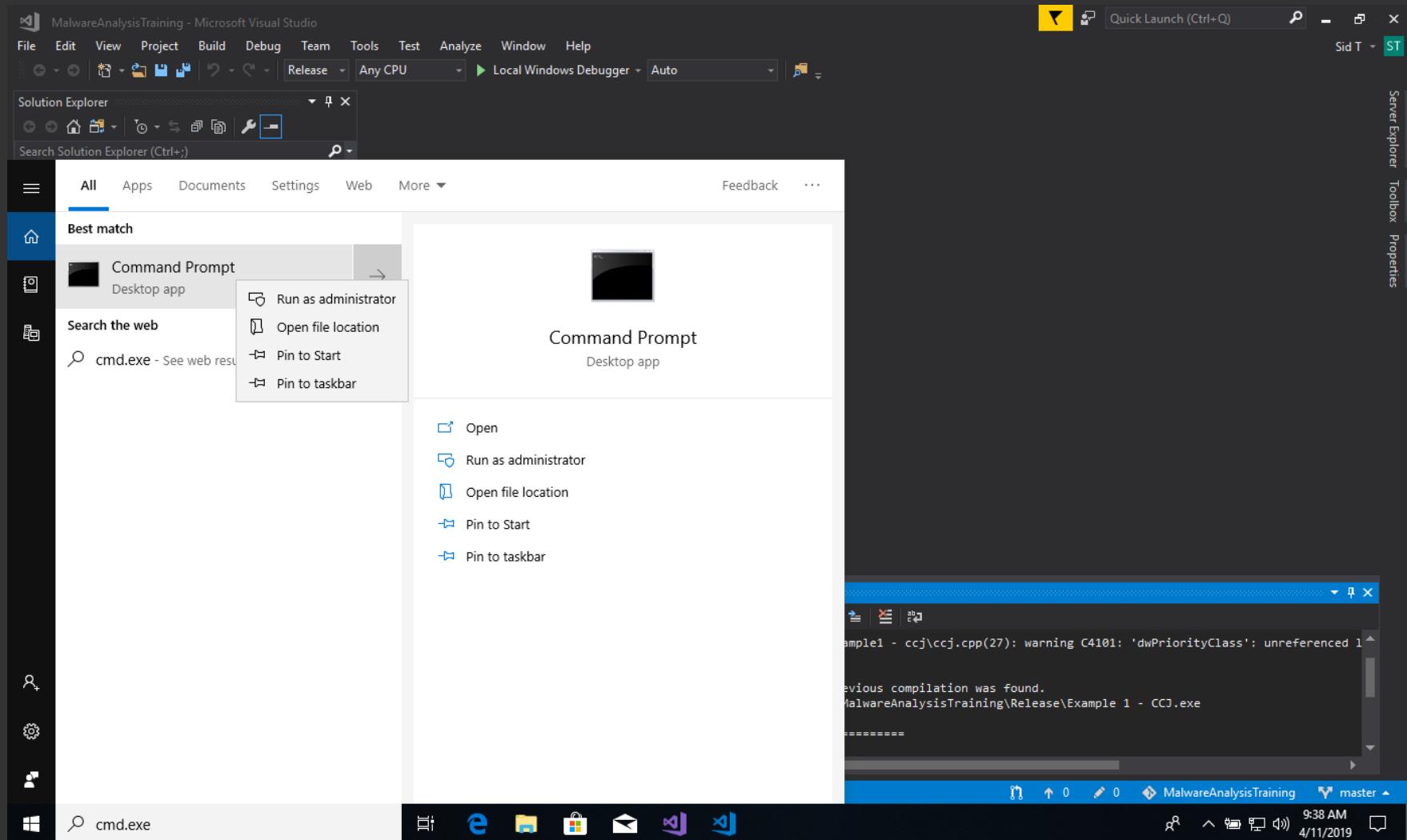
You should see success like the example below in the output window circled in red.



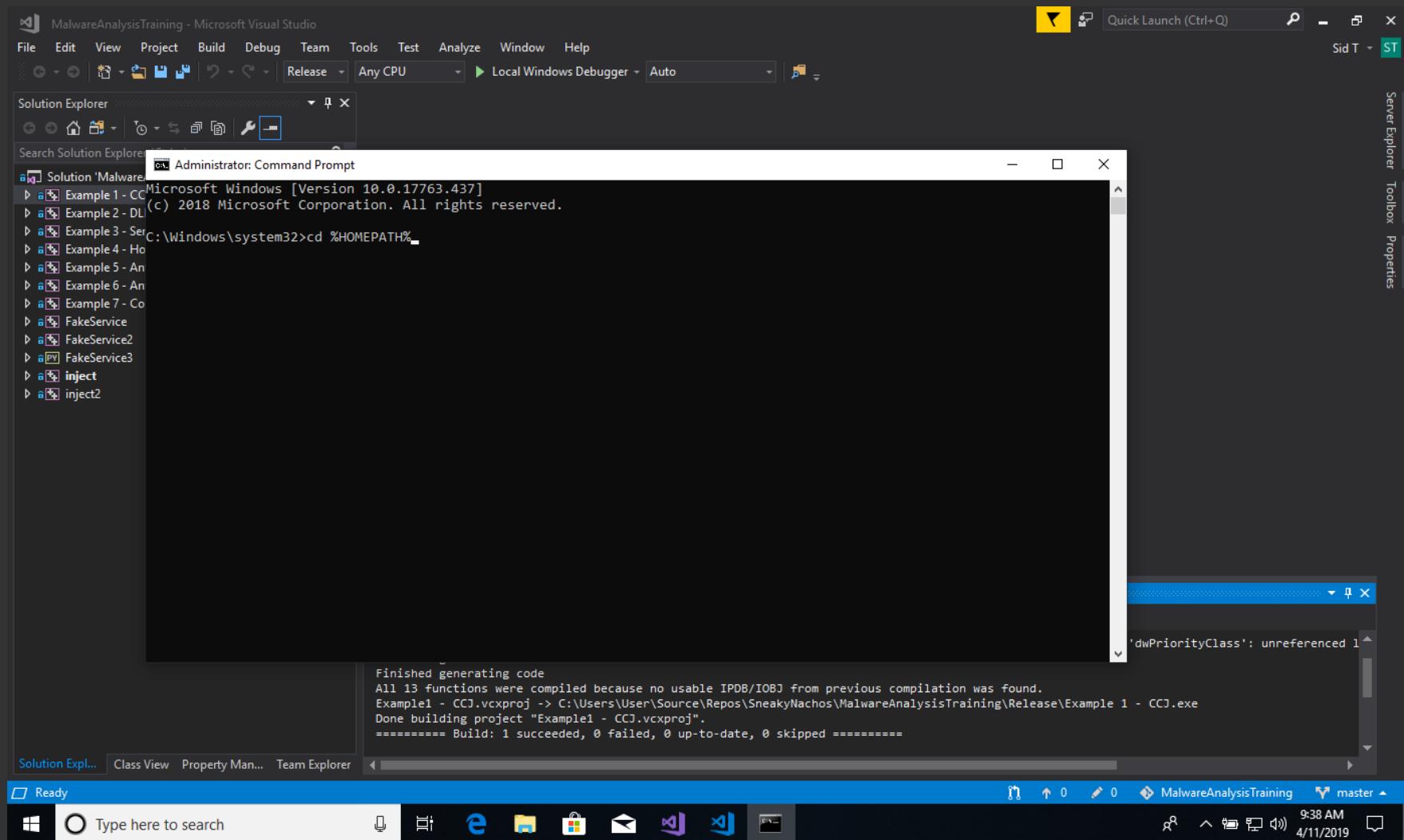
Let's go ahead and open up a command prompt running as Administrator.



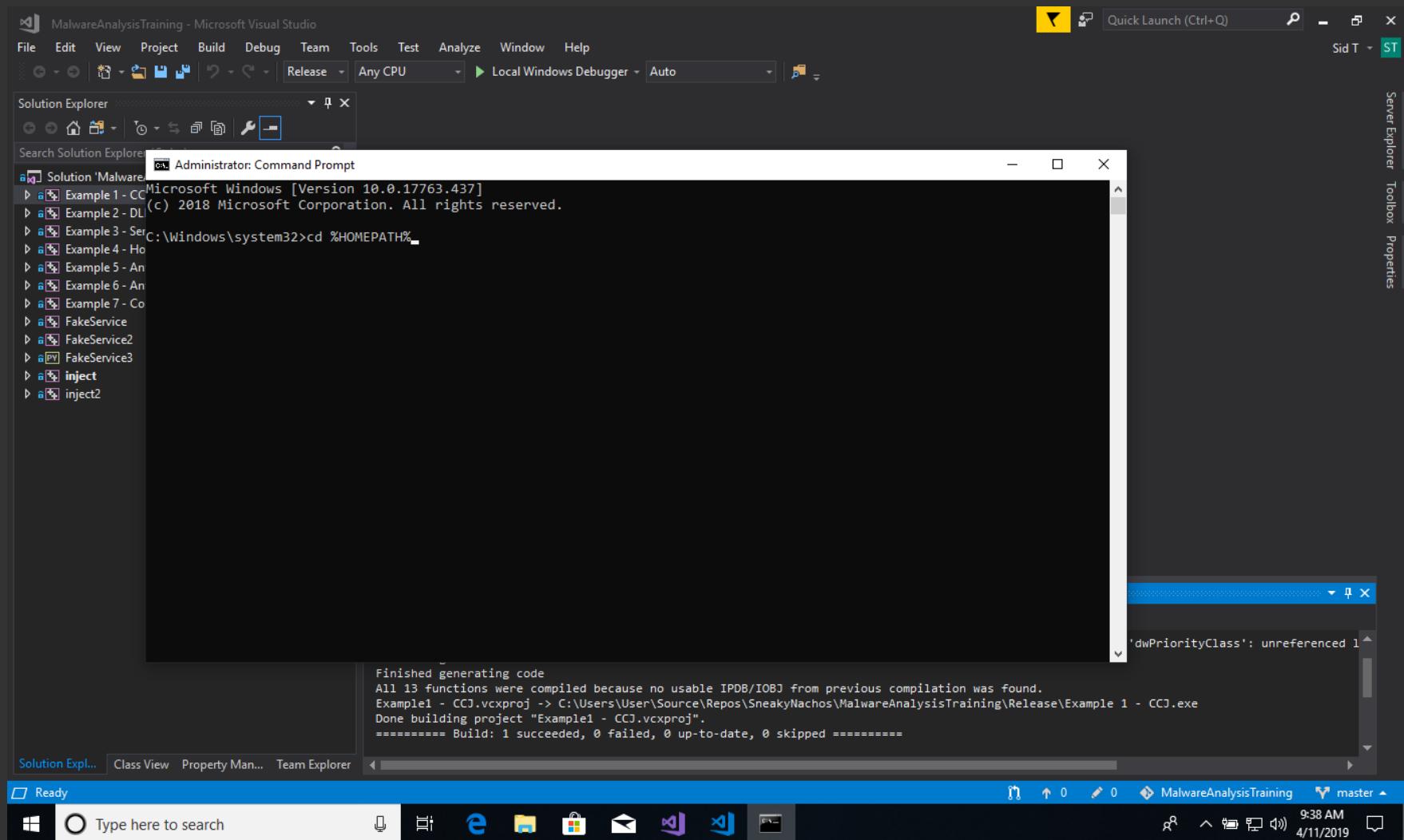
Remember to right click the command prompt to get the “Run as administrator” option.



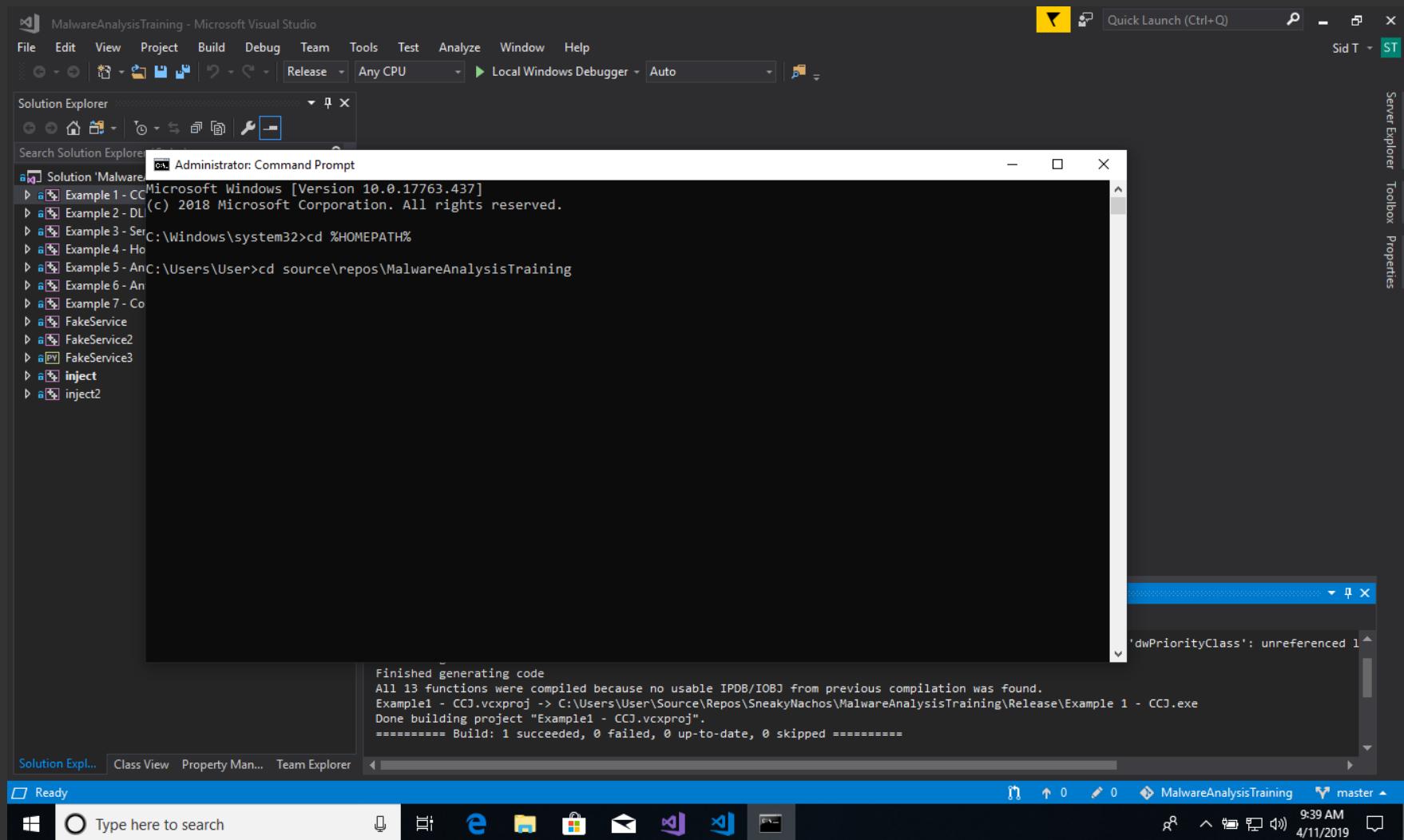
Now let us change to the directory of the project that you downloaded.



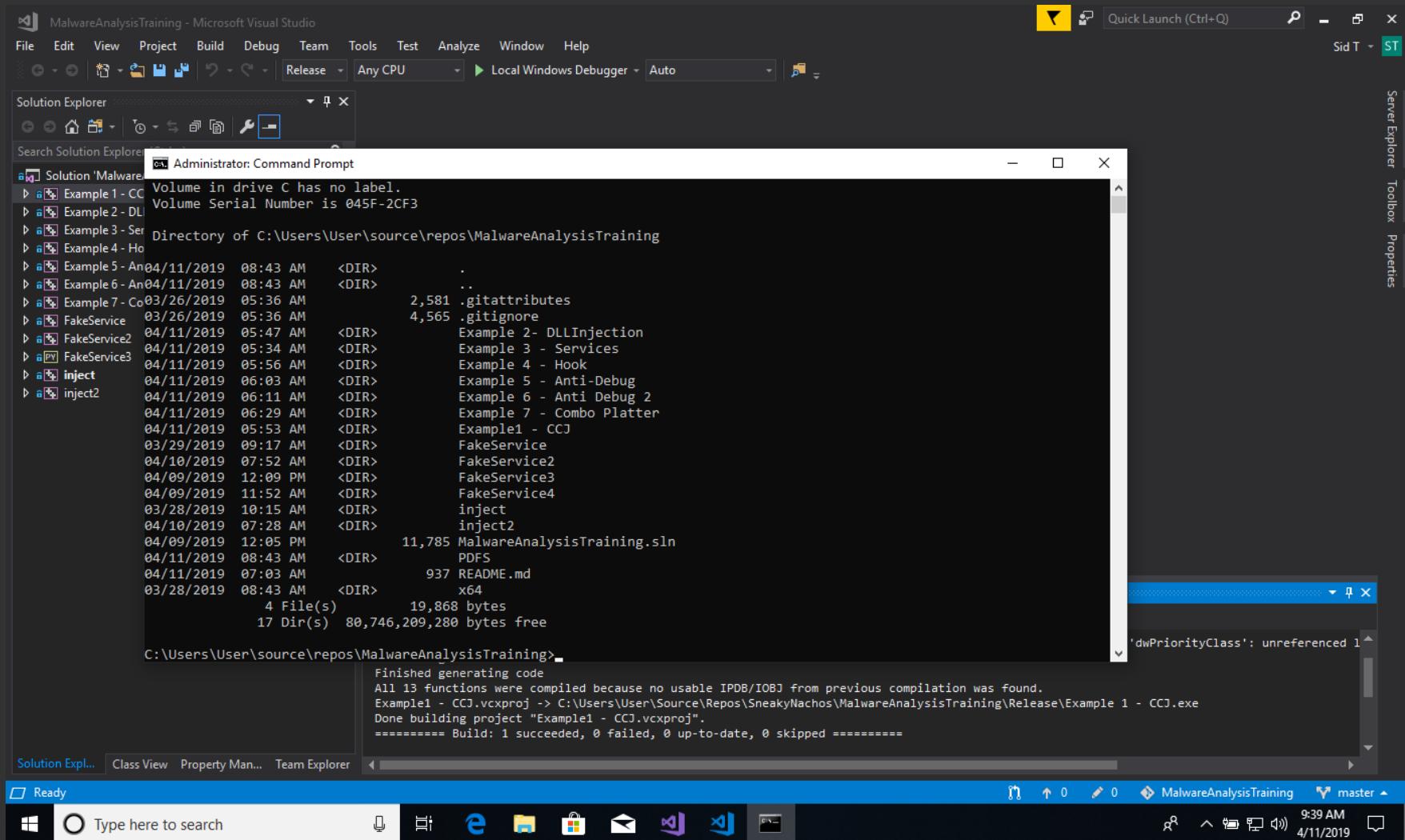
Now let us change to the directory of the project that you downloaded.



Note yours is probably on the Desktop, whereas the one built in the slides is from the source repo.



Now sitting in the highest level of the directory, after you've hit the "Build" button a new folder should show up. Either it will be the "x64" folder or the "Release" folder. If "x64", change into the "x64" directory and change into the "Release" folder within it. If you already have a "Release" folder at the top, go ahead and change into that directory.



When looking inside the “Release” folder, by typing “dir” you should see your example 1 binary built. Depending on when you downloaded the examples it could be “CCJ.exe” or “Example 1 – CCJ.exe”.

The screenshot shows a Microsoft Visual Studio interface with a terminal window open. The terminal window displays a command-line session in an Administrator Command Prompt. The user navigates to the directory C:\Users\User\source\repos\MalwareAnalysisTraining\x64 and runs a 'dir' command, which lists several subdirectories including 'Debug' and 'Release'. The user then changes to the 'Release' directory and runs another 'dir' command, showing files like 'CCJ.exe', 'CCJ.iobj', 'CCJ.ipdb', and 'CCJ.pdb'. A tooltip at the bottom right of the terminal window indicates an unreferenced variable named 'dwPriorityClass'. The Solution Explorer window on the left shows a solution named 'MalwareAnalysisTraining' containing multiple projects like 'Example 1 - CCJ', 'FakeService', and 'inject'. The status bar at the bottom shows the project name 'MalwareAnalysisTraining' and the branch 'master'.

```
C:\Users\User\source\repos\MalwareAnalysisTraining>cd x64
C:\Users\User\source\repos\MalwareAnalysisTraining\x64>dir
 Volume in drive C has no label.
 Volume Serial Number is 045F-2CF3

 Directory of C:\Users\User\source\repos\MalwareAnalysisTraining\x64

03/28/2019  08:43 AM    <DIR>      .
03/28/2019  08:43 AM    <DIR>      ..
04/11/2019  06:29 AM    <DIR>      Debug
04/10/2019  07:57 AM    <DIR>      Release
               0 File(s)   0 bytes
               4 Dir(s)  80,746,209,280 bytes free

C:\Users\User\source\repos\MalwareAnalysisTraining\x64>cd Release
C:\Users\User\source\repos\MalwareAnalysisTraining\x64\Release>dir
 Volume in drive C has no label.
 Volume Serial Number is 045F-2CF3

 Directory of C:\Users\User\source\repos\MalwareAnalysisTraining\x64\Release

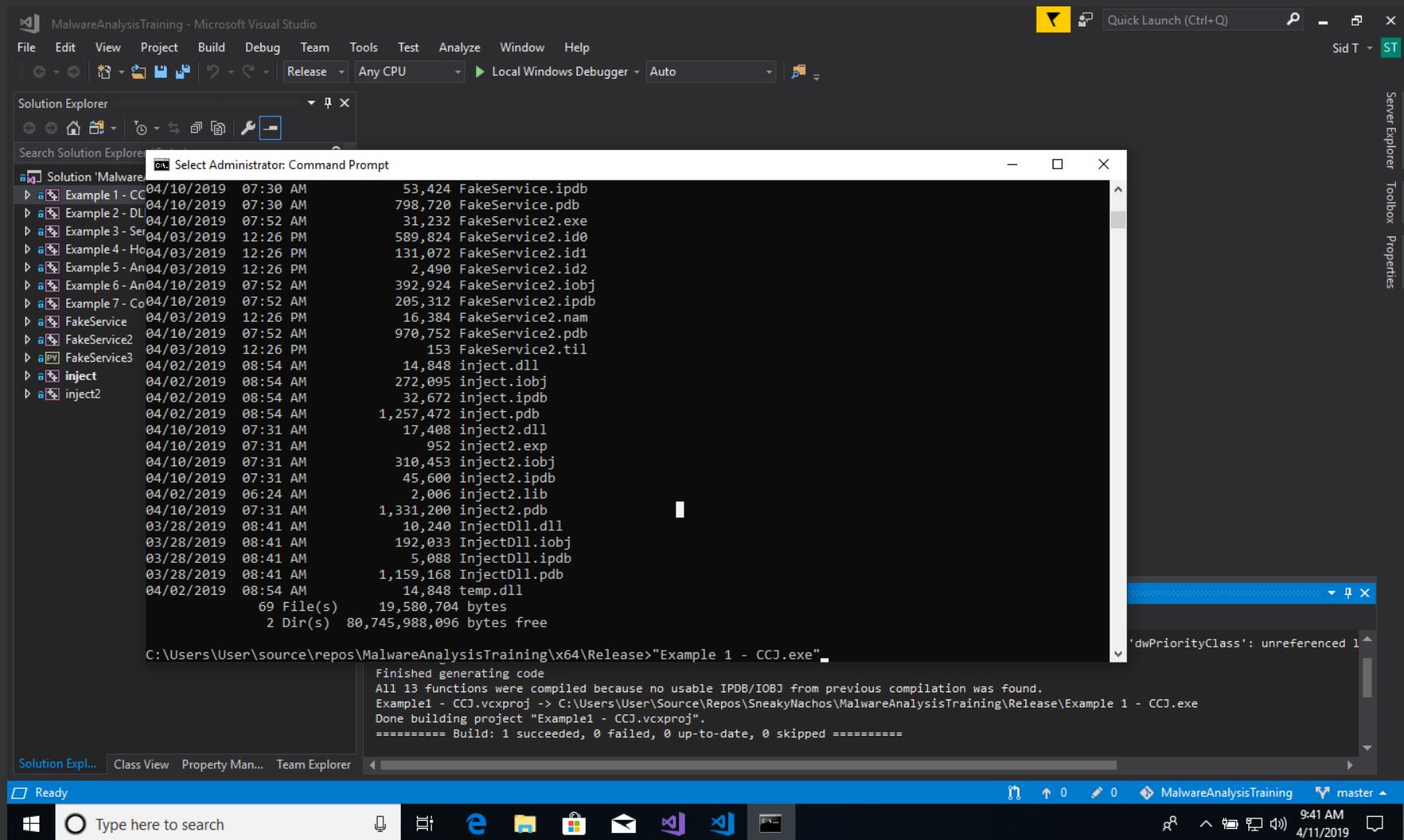
04/10/2019  07:57 AM    <DIR>      .
04/10/2019  07:57 AM    <DIR>      ..
03/26/2019  05:37 AM           15,872 CCJ.exe
03/26/2019  05:37 AM           70,552 CCJ.iobj
03/26/2019  05:37 AM           25,520 CCJ.ipdb
03/26/2019  05:37 AM           757,760 CCJ.pdb

Finished generating code
All 13 functions were compiled because no usable IPDB/IOBJ from previous compilation was found.
Example1 - CCJ.vcxproj -> C:\Users\User\Source\Repos\SneakyNachos\MalwareAnalysisTraining\Release\Example 1 - CCJ.exe
Done building project "Example1 - CCJ.vcxproj".
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ======
```

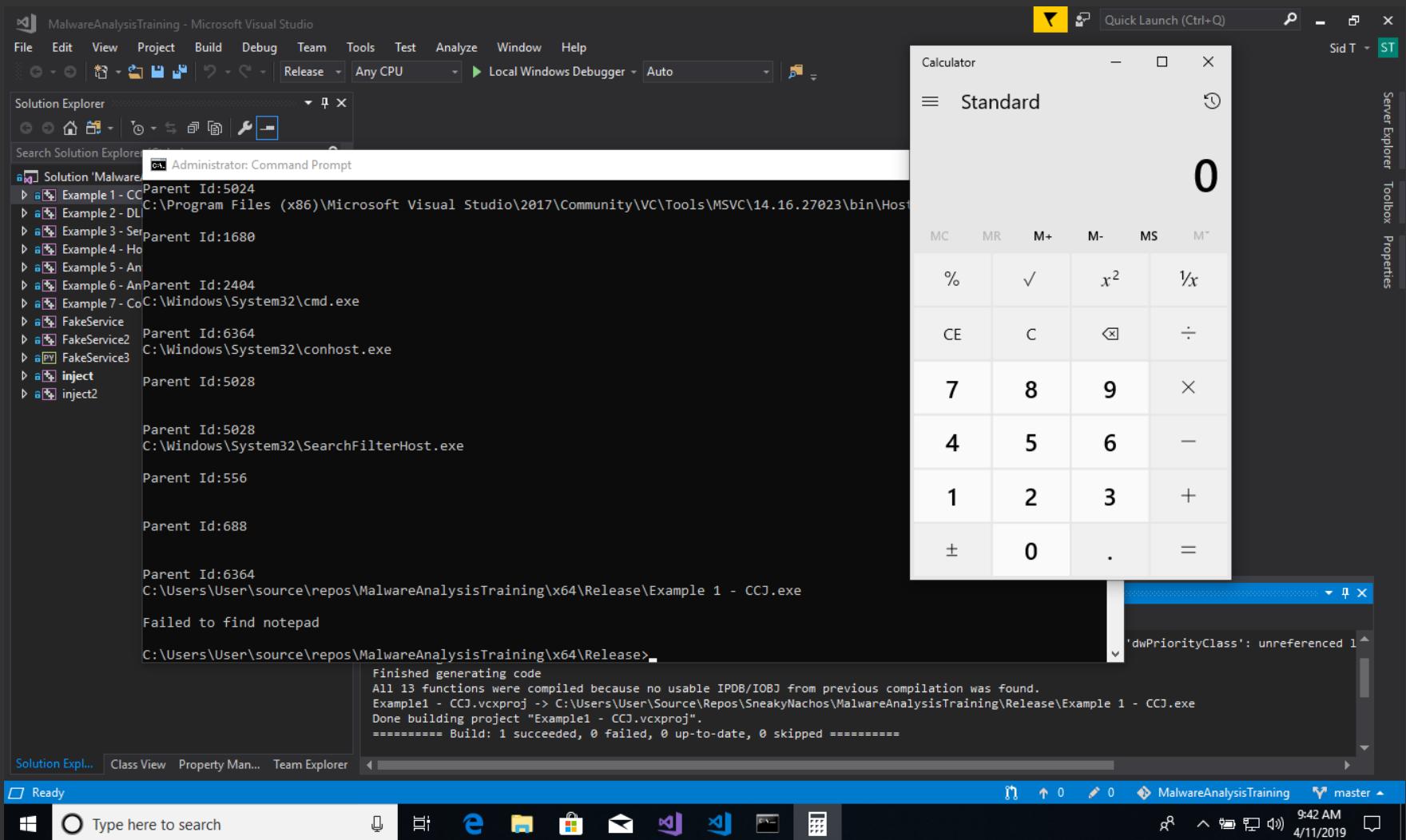
Common Issues

- The “Requirements.pdf” should assist you if you’re really stuck and out of options with compiling.
- Remember to delete the AppData/Local and AppData/Roaming for Visual Studio
- Make sure the “Any CPU” option is set to the correct target. “Any CPU” defaults to 32-bit which may be wrong.
- You may have not installed the C++ libraries for visual studio. (See Requirements.pdf)

Let's go ahead and do something dumb first. We'll just run the example first and see what the result is and define our initial timeline from that.



Looks like a number of things happened. First a bunch of process names were spat out. “Failed to find notepad” was printed, and a calculator spawned from nowhere. So at least now we have a general basis for the timeline.



Goals of Example 1

- You should be able to give me a general analysis and behavior timeframe of what the binary is doing.
- This includes the start all the way to end.

Starting the Initial Analysis

- The first step is a brief static analysis.
- Reasons for starting the static analysis first.
 - 1) Easily acquired meta data may give us insight into the behavior/purpose .
 - 2) Makes dynamic analysis faster.
 - 3) Reduce surface area of the analysis.

First Step - Strings

A “String” is a combination of bytes that is directly followed by a NULL byte, aka zero in memory.

As an example the word “CAT” in memory is actually encoded as “0x43 0x41 0x54 0x00 ”.

Where 0x43 is the hex encoded form of “C”, followed by “0x41” A, and “0x54” T. Which is then finished off by the “0x00” aka the NULL byte. We'll talk about memory arrangement later (Little endian/BigEndian).

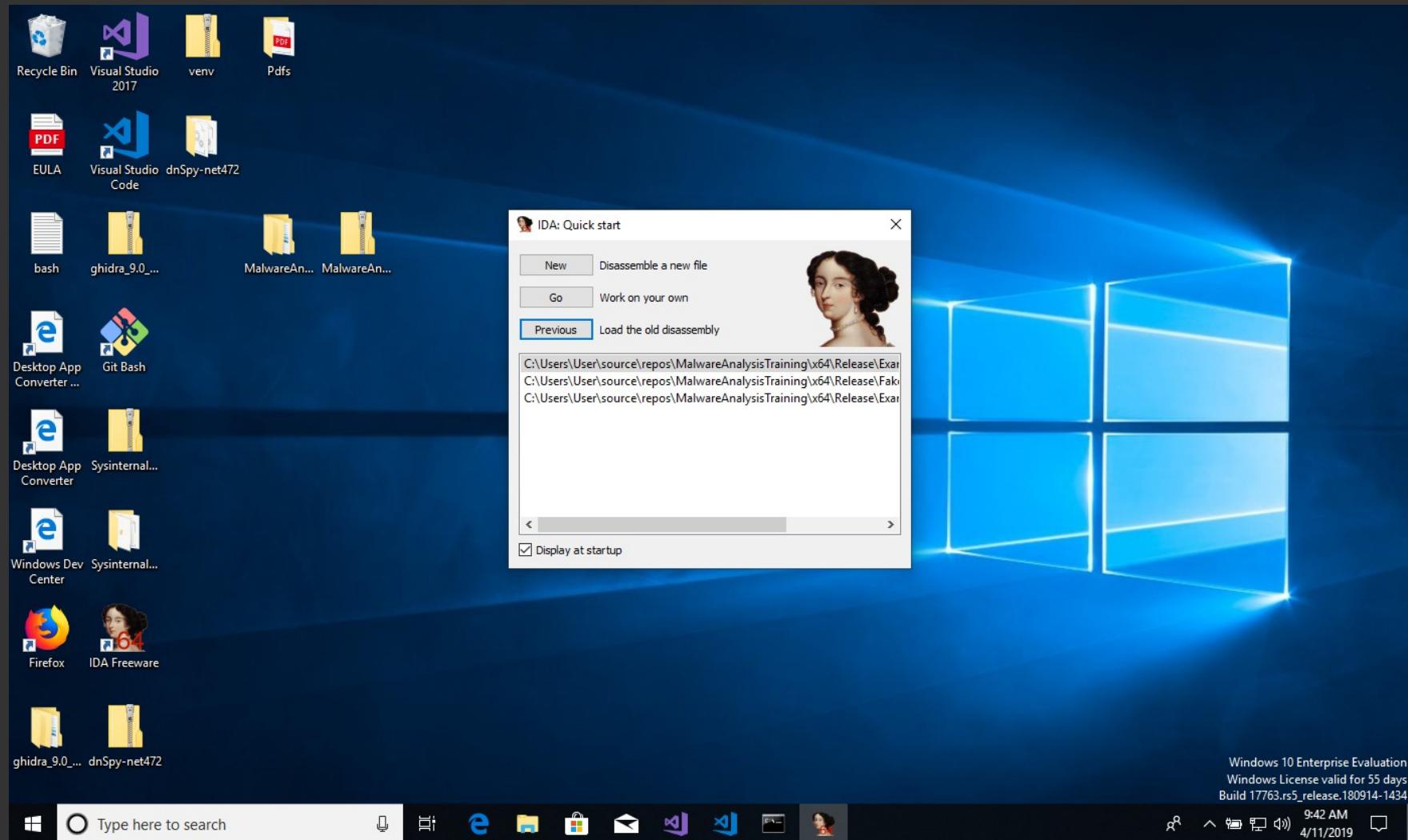
Why “Strings”?

Strings build a context of what the programmer wishes to do, is about to do, and possibly what didn't. Programmers use strings for printing, interacting with api's, debugging, web requests, error handling, file move/delete/create, data movement, and much more.

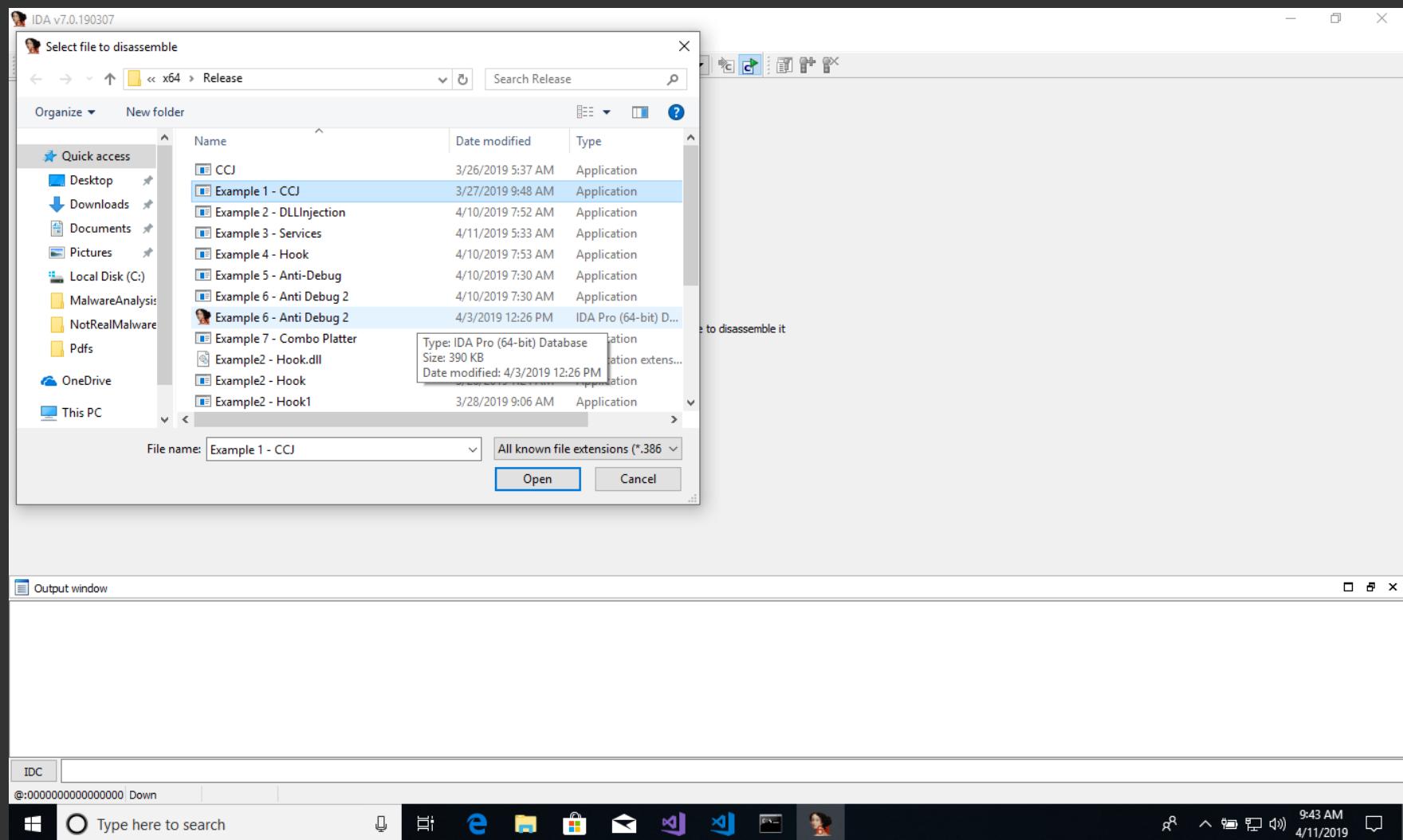
Why “Strings”?

This is very interesting to us as analysts because quite a number of the Windows API require strings to perform functionality. Thus if you can capture or find the string that is entering the Windows API and the Windows API function the malware wishes to make, you can generalize the action that the malware is trying to perform.

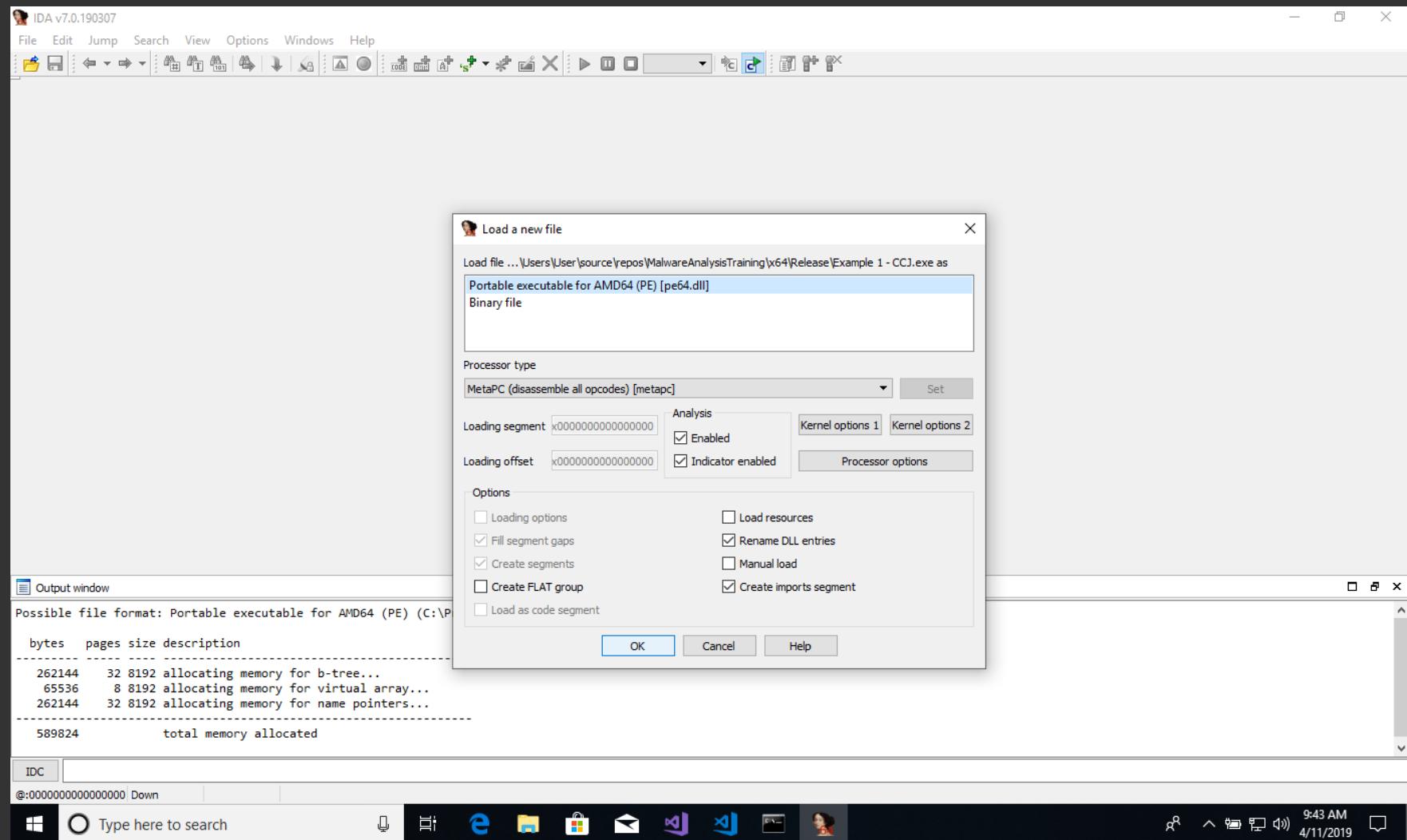
Now to look at the “Strings”, we'll pull up IDA Freeware and open the “.exe” in the Release folder.



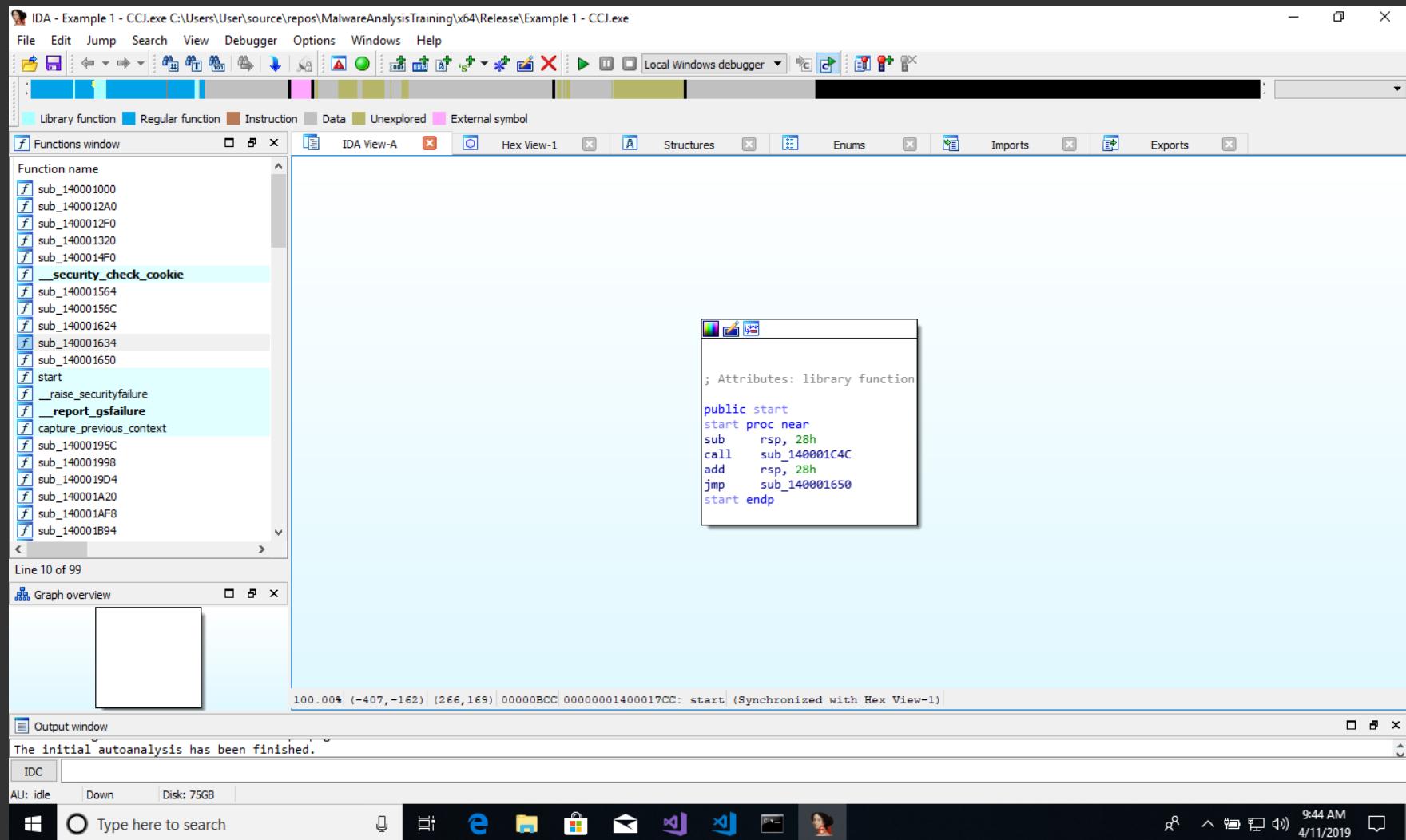
Open our “Example 1- CCJ” sitting in the Release folder.



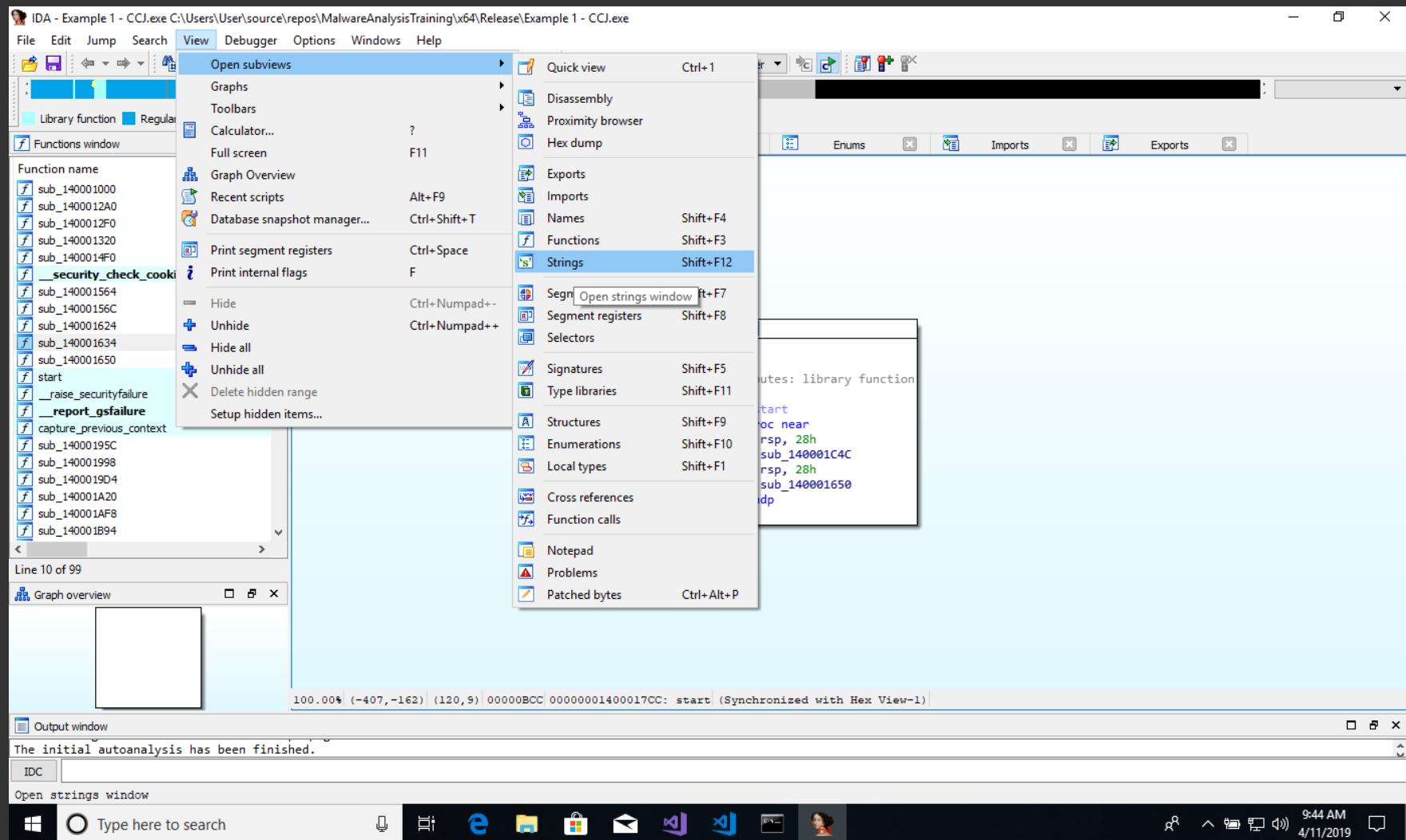
Don't worry about modifying anything when opening the file. All defaults are fine for this example.



This is what IDA should look like once you've successfully opened the example.



To open the strings sub-view, there's a tab at the top called “View”. Click the “View” drop down and hover over “Open subviews” and go down to “Strings” and click it.



A brand new sub-view should pop up, and we should now see all the strings in the example.

First things first. We want to note all interesting strings that we will to analyze first.

The screenshot shows the IDA Pro interface with the 'Strings window' open. The window displays a list of strings found in memory, categorized by type (C for character). The list includes various error messages and system-related strings. The 'Functions window' on the left shows several functions, with one named '_security_check_cookie' highlighted. The bottom status bar indicates 'The initial autoanalysis has been finished.'

Address	Length	Type	String
.rdata:0000...	00000012	C	Unknown exception
'\$'	000000F	C	bad allocation
'\$'	00000015	C	bad array new length
'\$'	000000B	C	Parent Id:
'\$'	0000007	C	Test!\n
'\$'	0000017	C	Failed to find notepad!
'\$'	000000F	C	Found notepad!
'\$'	0000015	C	Injection completed!
'\$'	0000007	C	Error!
'\$'	0000006	C	PAUSE
'\$'	0000005	C	GCTL
'\$'	0000009	C	.text\$mn
'\$'	000000C	C	.text\$mn\$00
'\$'	0000008	C	.text\$x
'\$'	0000009	C	.data\$5
'\$'	0000007	C	.00cfg
'\$'	0000009	C	.CRT\$XCA
'\$'	000000A	C	.CRT\$XCAA
'\$'	0000009	C	.CRT\$XCZ
'\$'	0000009	C	.CRT\$XIA
'\$'	000000A	C	.CRT\$XIAA
'\$'	000000A	C	.CRT\$XIAC
'\$'	0000009	C	.CRT\$XIZ
'\$'	0000009	C	.CRT\$XPA
'\$'	0000009	C	.CRT\$XPZ
'\$'	0000009	C	.CRT\$XTA
'\$'	0000009	C	.CRT\$XTZ
'\$'	0000007	C	.rdata
'\$'	0000009	C	.rdata\$r
'\$'	000000E	C	.rdata\$zzzdbq

I've highlighted in blue the ones that I found particularly interesting to this example. "Failed to find notepad" was what I considered the highest on my list because it printed out during the running of the binary from earlier.

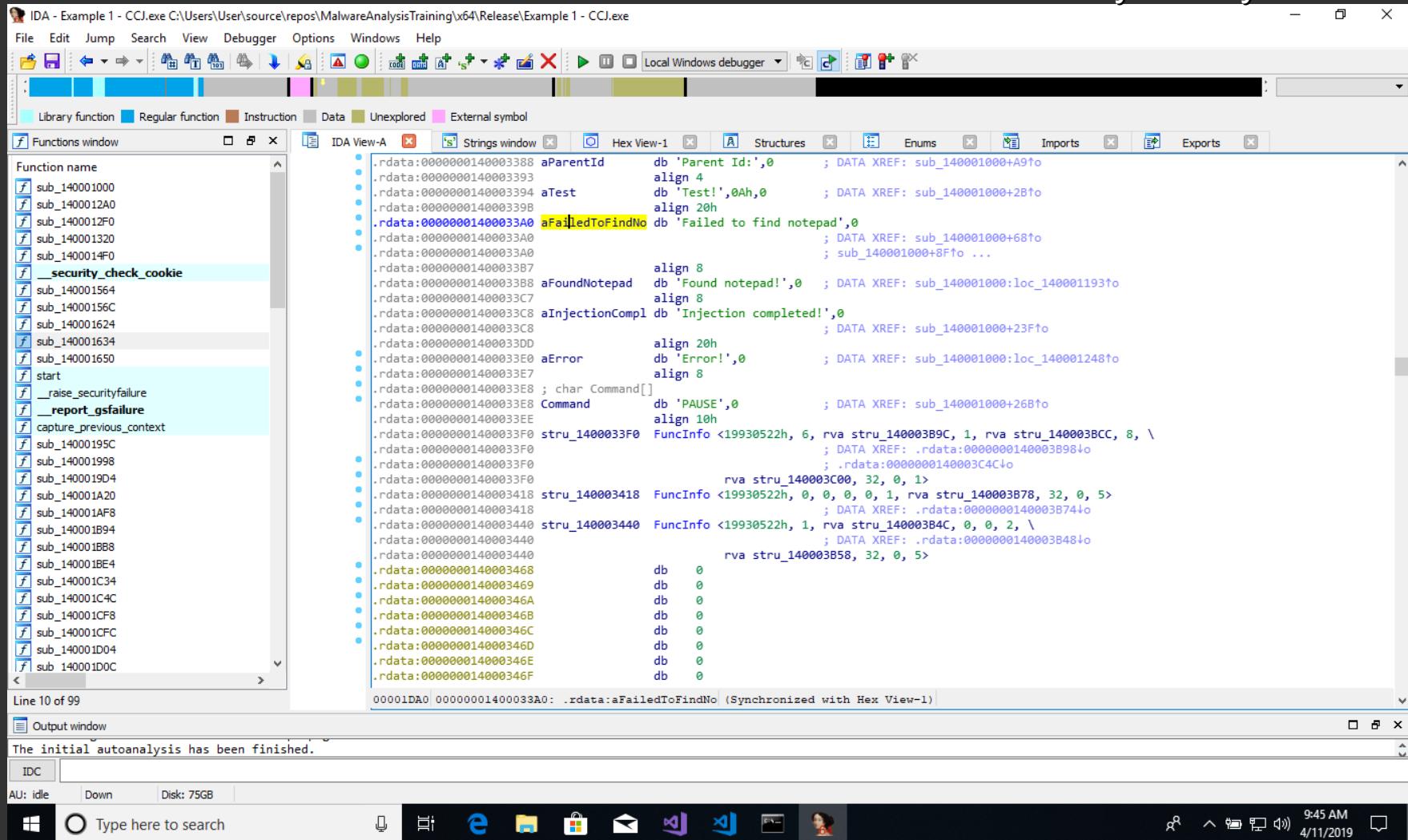
Let's double click "Failed to find notepad" now.

The screenshot shows the IDA Pro interface with the following details:

- Functions window:** On the left, it lists various functions, with `__security_check_cookie` currently selected.
- String table:** The main pane displays a table of strings, with several entries highlighted in blue:
 - Parent Id:
 - Test!\n
 - Failed to find notepad
 - Found notepad!
 - Injection completed!
 - Error!
 - PAUSE
 - GCTL
 - .text\$mn
 - .text\$mn\$00
 - .text\$x
 - .data\$5
 - .00cfg
 - .CRT\$XCA
 - .CRT\$XCAA
 - .CRT\$XCZ
 - .CRT\$XIA
 - .CRT\$XIAA
 - .CRT\$XIAC
 - .CRT\$KIZ
 - .CRT\$XPA
 - .CRT\$XPZ
 - .CRT\$XTA
 - .CRT\$XTZ
 - .rdata
 - .rdata\$r
 - .rdata\$zzzdbq
- Graph overview:** A small window at the bottom left showing a graph structure.
- Output window:** At the bottom, it says "The initial autoanalysis has been finished."
- Taskbar:** Shows the Windows Start button, a search bar, and pinned icons for File Explorer, Edge, File History, Mail, and File Explorer.
- System tray:** Shows the date and time as 9:45 AM, 4/11/2019.

You'll be taken to this section if you double clicked the "Failed to find notepad" string. This is the .rdata section and is filled with read only data. IDA gave database names to the different strings. I highlighted to the right of the string that database name.

Now click the "aFailedToFindNo" name and hit the "X" button on your keyboard.

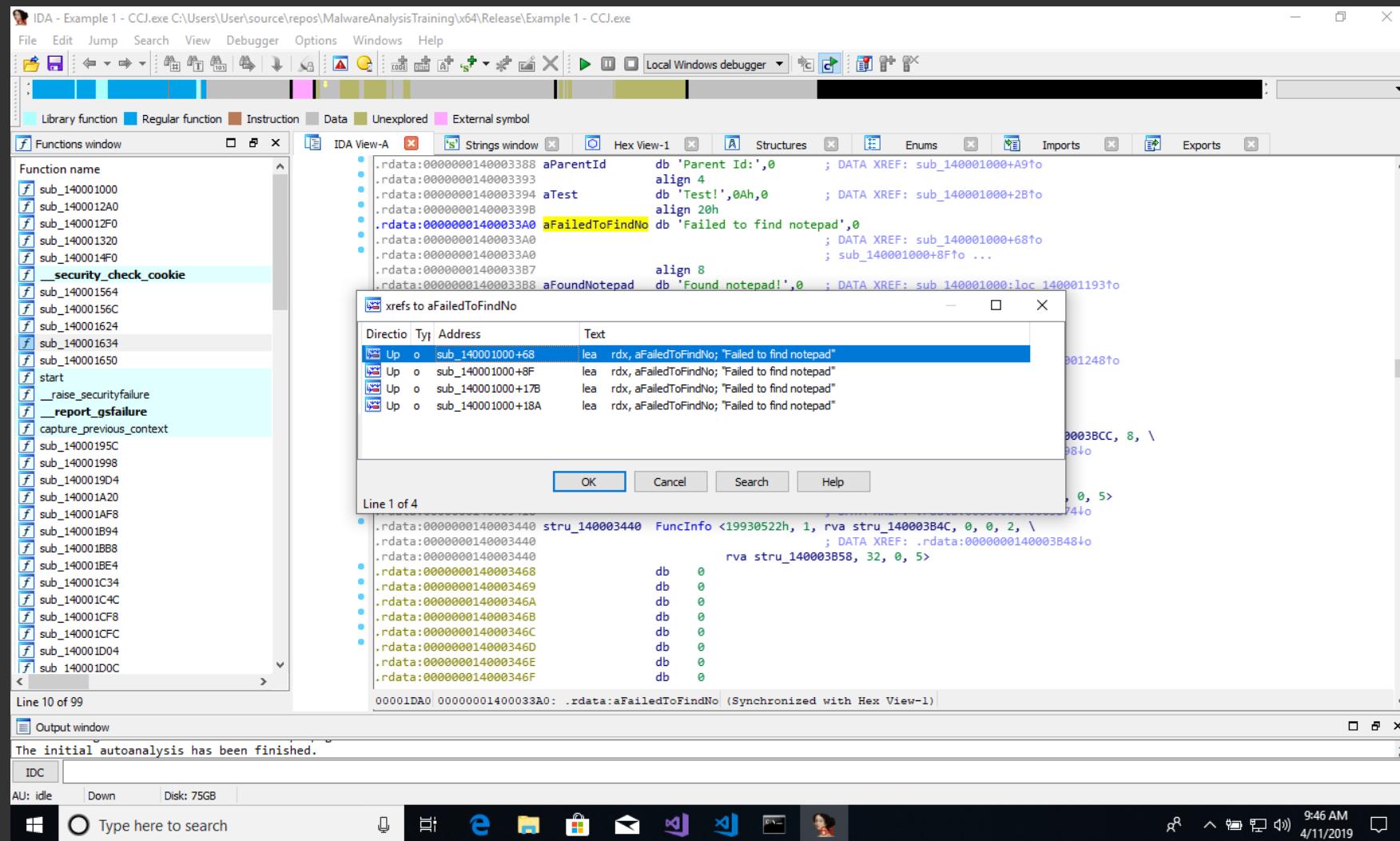


The screenshot shows the IDA Pro interface with the following details:

- Title Bar:** IDA - Example 1 - CCJ.exe C:\Users\User\source\repos\MalwareAnalysisTraining\x64\Release\Example 1 - CCJ.exe
- Menu Bar:** File, Edit, Jump, Search, View, Debugger, Options, Windows, Help
- Toolbars:** Standard toolbar with icons for file operations, search, and debugger.
- Function Window:** Shows a list of functions, with `__security_check_cookie` currently selected.
- IDA View-A:** Assembly view showing the .rdata section. A specific string, `aFailedToFindNo`, is highlighted in yellow.
- Strings Window:** Shows the string `'Failed to find notepad'`.
- Hex View-1:** Hex dump view corresponding to the assembly code.
- Imports:** Import table view.
- Exports:** Export table view.
- Output Window:** Displays the message: "The initial autoanalysis has been finished."
- Status Bar:** Shows "AU: idle", "Disk: 75GB", and the system clock "9:45 AM 4/11/2019".

Hitting the “X” on the IDA database name should bring up the cross references windows. This allows us to track in the assembly where the string “Failed to find notepad” is being used.

Let's double click the first option that I highlighted in blue below.



After double clicking the first option we'll be taken to the location of where in the assembly the string is used. I circled the location in red below.

Let's drag ourselves to the top of the assembly and record where we're at.

The screenshot shows the IDA Pro interface with the assembly view open. A red circle highlights the instruction at address `loc_14000119A`:

```
mov    rbx, rax
lea    rdx, aFailedToFindNo ; "Failed to find notepad"
jmp    loc_14000119A
```

Below this instruction, the assembly code continues:

```
loc_14000119A:
mov    rcx, cs:@cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::basic_ostream<char, std::char_traits<char> >
call   sub_140001320
lea    rdx, sub_1400014F0
mov    rcx, rax
call   cs:@?6?$basic_ostream@DU?$char_traits@D@std@@std@@QEAACEAV01@P6AAEAV01@AEAV01@Z@Z ; std::basic_ostream<char, std::char_traits<char> >
xor   edx, edx ; lpAddress
mov    [rsp+2A8h+f1Protect], 40h ; f1Protect
mov    r9d, 3000h ; f1AllocationType
mov    r8d, 102h ; dwSize
mov    rcx, rbx ; hProcess
call   cs:VirtualAllocEx
xor   esi, esi
```

The status bar at the bottom indicates the current assembly address is `sub_140001000+68`.

After dragging ourselves to the top block, we can see our name highlighted in yellow below. I also drew a circles around it. IDA tends to name locations with "sub_" and then the offset of where it is in the file.

We'll keep this noted for now in our notes.

The screenshot shows the IDA Pro interface with the title bar "IDA - Example 1 - CCJ.exe C:\Users\User\source/repos\MalwareAnalysisTraining\x64\Release\Example 1 - CCJ.exe". The left pane displays the "Functions window" containing a list of functions, with "sub_140001000" highlighted. The right pane shows the assembly code:

```
; Section size in file : 00001600 ( 5632.)
; Offset to raw data for section: 00000400
; Flags 6000020: Text Executable Readable
; Alignment : default
; PDB File Name : C:\Users\User\source/repos\MalwareAnalysisTraining\x64\Release\Example 1 - CCJ.pdb

.686p
.mmx
.model flat

; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use64

org 14000100h
assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing

sub_140001000 proc near

f1Protect= dword ptr -28h
dwCreationFlags= dword ptr -20h
lpThreadId= qword ptr -18h
dwSize= dword ptr -28h
SubStr= byte ptr -20h
var_18= qword ptr -18h
var_8= byte ptr -8
arg_0= qword ptr 8
arg_8= qword ptr 10h

mov    [rsp+arg_0], rbx
mov    [rsp+arg_8], rsi
push   rdi
```

A red oval highlights the start of the function definition "sub_140001000 proc near". The status bar at the bottom indicates "100.00% (80,283) (422,386) 00000400 0000000140001000: sub_140001000 (Synchronized with Hex View-1)". The bottom taskbar shows the Windows Start button, a search bar, and various pinned icons like File Explorer, Edge, Mail, and File History.

Let's go back to strings and click another string like "Found notepad!". Remember to hit "X" on the highlighted IDA name below so that the cross references will work.

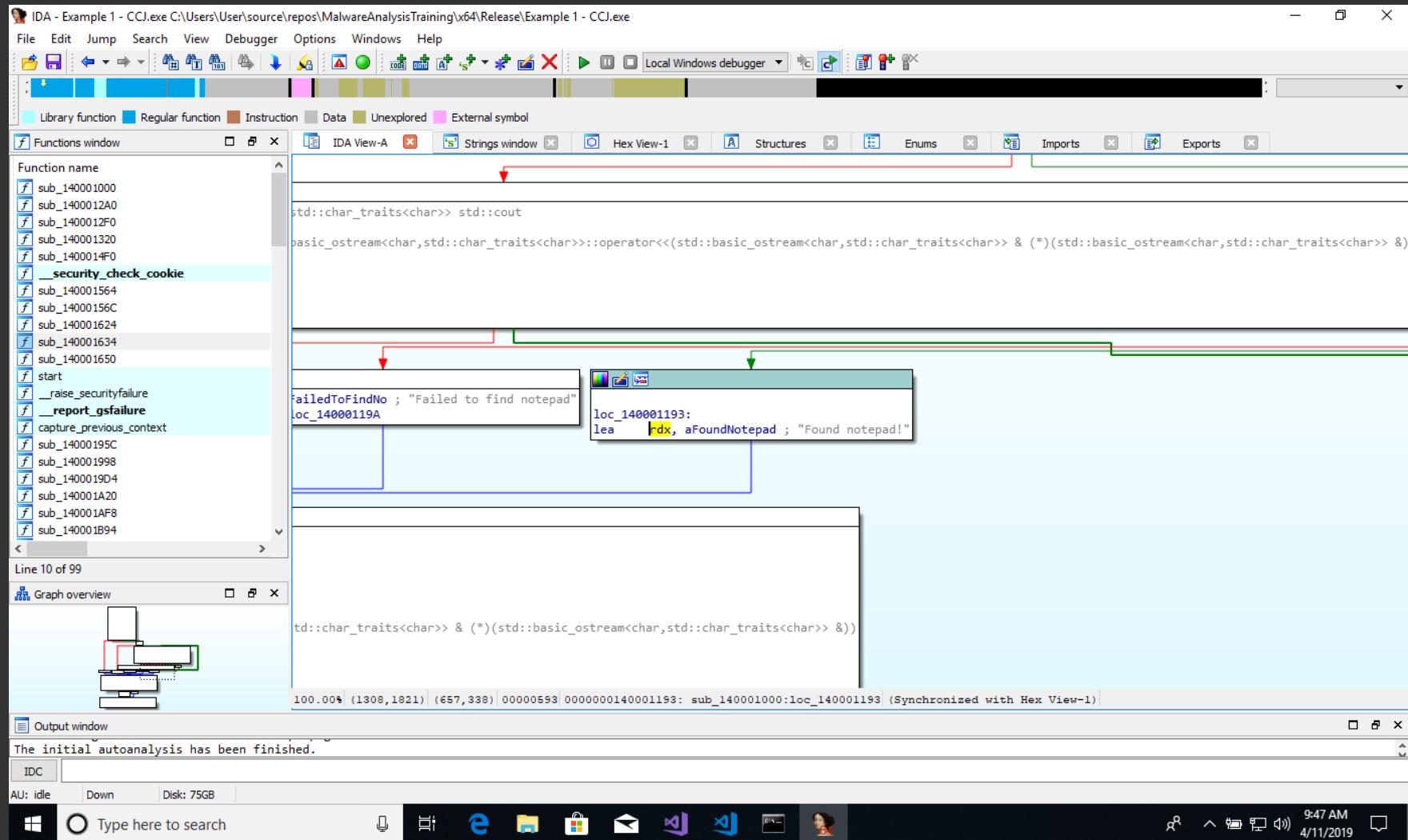
The screenshot shows the IDA Pro interface with the following details:

- Title Bar:** IDA - Example 1 - CCJ.exe C:\Users\User\source/repos\MalwareAnalysisTraining\x64\Release\Example 1 - CCJ.exe
- Menu Bar:** File, Edit, Jump, Search, View, Debugger, Options, Windows, Help
- Toolbars:** Standard toolbar with icons for file operations, search, and debugger.
- Function List:** Functions window on the left, listing various functions such as `sub_140001000`, `sub_1400012A0`, `sub_1400012F0`, `sub_140001320`, `sub_1400014F0`, `__security_check_cookie`, `sub_140001564`, `sub_14000156C`, `sub_140001624`, `sub_140001634`, `sub_140001650`, `start`, `__raise_securityfailure`, `__report_gsfailure`, `capture_previous_context`, `sub_14000195C`, `sub_140001998`, `sub_1400019D4`, `sub_140001A20`, `sub_140001AF8`, `sub_140001B94`, `sub_140001BB8`, `sub_140001BE4`, `sub_140001C34`, `sub_140001C4C`, `sub_140001CF8`, `sub_140001CF0`, `sub_140001D04`, and `sub_140001D0C`.
- Assembly View:** The main pane displays assembly code. A specific string, `aFoundNotePad`, is highlighted with a red circle. The assembly code includes:

```
.rdata:000001400033A0 aFailedToFindNo db 'Failed to find notepad',0
.rdata:000001400033A0 align 8
.rdata:000001400033A0 db 'Found notepad!',0
.rdata:00000140003387 align 8
.rdata:00000140003388 aInjectionCompl db 'Injection completed!',0
.rdata:000001400033C8 align 20h
.rdata:000001400033E0 error db 'Error!',0
.rdata:000001400033E7 align 8
.rdata:000001400033E8 ; char Command[]
.rdata:000001400033E8 Command db 'PAUSE',0
.rdata:000001400033EE align 10h
.rdata:000001400033F0 stru_1400033F0 FuncInfo <19930522h, 6, rva stru_140003B9C, 1, rva stru_140003BCC, 8, \
.rdata:000001400033F0 ; DATA XREF: .rdata:000000140003B98!o
.rdata:000001400033F0 ; .rdata:000000140003C4C!o
.rdata:000001400033F0 rva stru_140003C00, 32, 0, 1>
.rdata:00000140003418 stru_140003418 FuncInfo <19930522h, 0, 0, 0, 1, rva stru_140003B78, 32, 0, 5>
.rdata:00000140003418 ; DATA XREF: .rdata:000000140003B74!o
.rdata:00000140003440 stru_140003440 FuncInfo <19930522h, 1, rva stru_140003B4C, 0, 0, 2, \
.rdata:00000140003440 ; DATA XREF: .rdata:000000140003B48!o
.rdata:00000140003440 rva stru_140003B58, 32, 0, 5>
.rdata:00000140003468 db 0
.rdata:00000140003469 db 0
.rdata:0000014000346A db 0
.rdata:0000014000346B db 0
.rdata:0000014000346C db 0
.rdata:0000014000346D db 0
.rdata:0000014000346E db 0
.rdata:0000014000346F db 0
.rdata:00000140003470 ; Debug Directory entries
.rdata:00000140003470 dd 0
.rdata:00000140003474 dd 5C9BA948h
.rdata:00000140003478 dw 0
.rdata:00000140003478 ; Characteristics
.rdata:00000140003478 ; TimeDateStamp: Wed Mar 27 16:48:08 2019
.rdata:00000140003478 dw 0
.rdata:00000140003478 ; MajorVersion
```
- Hex View:** Below the assembly view, there is a hex dump of the memory starting at address `000001DB8`.
- Status Bar:** Shows "AU: idle", "Disk: 75GB", and the current date and time: "9:47 AM 4/11/2019".
- Taskbar:** Shows the Windows Start button, a search bar, and pinned icons for File Explorer, Edge, File History, Mail, and Task View.

And now we're taken back to the place where "Found notepad!" is used in the assembly.

Let's drag ourselves back up to the top.



Looks like it's within the same place that the failed to find notepad message also resides.

The screenshot shows the IDA Pro interface with the assembly view open. The assembly code for the function `sub_140001000` is displayed. The code includes comments indicating it is a pure code segment with executable permissions, and it uses the `CODE` segment. It also includes assumptions about CPU features like `.686p`, `.mmx`, and `.model flat`. The assembly code itself starts with a proc near directive and defines several local variables (flProtect, dwCreationFlags, lpThreadId, pe, dwSize, SubStr, var_18, var_8, arg_0, arg_8) using the `dwSize` register. The assembly code then contains standard x86 instructions like `mov` and `push`.

```
; Section size in file : 00001600 ( 5632.)
; Offset to raw data for section: 00000400
; Flags 6000020: Text Executable Readable
; Alignment    : default
; PDB File Name : C:\Users\User\source/repos\MalwareAnalysisTraining\x64\Release\Example 1 - CCJ.pdb

.686p
.mmx
.model flat

; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use64
assume cs:_text
;org 14000100h
assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing

sub_140001000 proc near

flProtect= dword ptr -288h
dwCreationFlags= dword ptr -280h
lpThreadId= qword ptr -278h
pe= PROCESSENTRY32W ptr -268h
dwSize= dword ptr -28h
SubStr= byte ptr -20h
var_18= qword ptr -18h
var_8= byte ptr -8
arg_0= qword ptr 8
arg_8= qword ptr 10h

    mov    [rsp+arg_0], rbx
    mov    [rsp+arg_8], rsi
    push   rdi

100.00% (80,283) (422,386) 00000400 0000000140001000: sub_140001000 (Synchronized with Hex View-1)
```

The bottom status bar indicates "The initial autoanalysis has been finished." and shows system information like "AU: idle", "Disk: 75GB", and the date/time "4/11/2019 9:46 AM".

Step 2 - Imports

- In Windows for the programmer to actually interact and perform functionality on the system they must go through the Windows API. Every action performed that creates/modifies/deletes anything on the system must go through this tunnel. This information in most cases, unless its brought in at runtime, is available via the “Imports” section of the binary.

What should we look for?

- Our goal should be when analyzing the imports is to find interesting imports that shine light on the purpose and behavior of the malware.
- Read/Write of memory
- File movement/deletion/creation
- Registry creation/edits/deletion
- The list goes on...

Let's bring up the imports tab and have a look around.

I've highlighted in blue the function I found most interesting at the start.

CreateToolhelp32Snapshot

The screenshot shows the IDA Pro interface with the 'Imports' tab highlighted by a red box. The imports table lists various Windows API functions from the KERNEL32 library. The function 'CreateToolhelp32Snapshot' is highlighted in blue in the Functions window on the left. The bottom status bar indicates 'The initial autoanalysis has been finished.'

Address	Ordinal	Name	Library
00000000		CreateToolhelp32Snapshot	KERNEL32
00000000		Process32FirstW	KERNEL32
00000000		CloseHandle	KERNEL32
00000000		OpenProcess	KERNEL32
00000000		QueryFullProcessImageNameA	KERNEL32
00000000		Process32NextW	KERNEL32
00000000		VirtualAllocEx	KERNEL32
00000000		WriteProcessMemory	KERNEL32
00000000		CreateRemoteThread	KERNEL32
00000000		WaitForSingleObject	KERNEL32
00000000		IsDebuggerPresent	KERNEL32
00000000		InitializeSLListHead	KERNEL32
00000000		GetSystemTimeAsFileTime	KERNEL32
00000000		GetCurrentThreadId	KERNEL32
00000000		GetCurrentProcessId	KERNEL32
00000000		QueryPerformanceCounter	KERNEL32
00000000		IsProcessorFeaturePresent	KERNEL32
00000000		TerminateProcess	KERNEL32
00000000		GetCurrentProcess	KERNEL32
00000000		SetUnhandledExceptionFilter	KERNEL32
00000000		UnhandledExceptionFilter	KERNEL32
00000000		RtlVirtualUnwind	KERNEL32
00000000		RtlLookupFunctionEntry	KERNEL32
00000000		RtlCaptureContext	KERNEL32
00000000		GetModuleHandleW	KERNEL32
00000000		std::basic_ostream<char, std::char_traits<char>>::operator<<	MSVCP140
00000000		std::basic_ios<char, std::char_traits<char>>::setstate(i)	MSVCP140
00000000		std::basic_ostream<char, std::char_traits<char>>::flush	MSVCP140
00000000		std::basic_ostream<char, std::char_traits<char>>::Osflush	MSVCP140
00000000		std::basic_streambuf<char, std::char_traits<char>>::sp	MSVCP140

Why CreateToolhelp32Snapshot?

- Allows one to snapshot a list of all processes running on the system.
- We saw at the start when running our binary, a bunch of processes being displayed in our command prompt.
- We can probably deduce that this imported function must be the one allowing the binary to get this information.

Double click the CreateToolhelp32Snapshot and you'll be greeted with this screen. If you press "X" on the pink version of the CreateToolhelp32Snapshot you'll bring up the cross references for this imported function in the assembly.

The screenshot shows the IDA Pro interface with the following details:

- Title Bar:** IDA - Example 1 - CCJ.exe C:\Users\User\source/repos\MalwareAnalysisTraining\x64\Release\Example 1 - CCJ.exe
- Menu Bar:** File, Edit, Jump, Search, View, Debugger, Options, Windows, Help
- Toolbar:** Includes icons for file operations, search, and debugger.
- Function List:** Functions window on the left lists various functions, with `_security_check_cookie` highlighted.
- Assembly View:** The main pane displays assembly code. A specific line of code is highlighted in yellow:

```
.idata:0000000140003000 ; =====
.idata:0000000140003000 ; Segment type: Externs
.idata:0000000140003000 ; _ida
.idata:0000000140003000 ; HANDLE __stdcall CreateToolhelp32Snapshot(DWORD dwFlags, DWORD th32ProcessID)
.idata:0000000140003000     extrn CreateToolhelp32Snapshot:qword
```
- Output Window:** Shows the message: "The initial autoanalysis has been finished."
- System Taskbar:** Shows the Windows Start button, a search bar, and several pinned application icons (File Explorer, Edge, File History, Mail, File Explorer, File Explorer, File Explorer).
- System Clock:** Displays the time as 9:49 AM and the date as 4/11/2019.

Looking at the cross references (xrefs). We can see where it's located in the assembly.

Circles below is also where the address is relative to the IDA name for the function. Recognize this place? You should, Failed to find notepad is there.

The screenshot shows the IDA Pro interface with the assembly view open. The assembly code for the function `sub_140001000` is displayed, showing several external symbols like `CreateToolhelp32Snapshot` and `VirtualAllocEx`. A red circle highlights a context menu or a cross-reference dialog box that has popped up over the assembly code. This dialog box lists three entries: "Up p sub_140001000+56", "Up r sub_140001000+56", and "D... o .idata:0000000140003E3C". The entry "Up p sub_140001000+56" is selected. The assembly code below the dialog shows the `call cs:CreateToolhelp32Snapshot` instruction.

```
File Edit Jump Search View Debugger Options Windows Help
File View Jump Search Debugger Options Windows Help
Library function Regular function Instruction Data Unexplored External symbol
Functions window IDA View-A Strings window Hex View-1 Structures Enums Imports Exports
Function name
sub_140001000
sub_1400012A0
sub_1400012F0
sub_140001320
sub_1400014F0
__security_check_cookie
sub_140001564
sub_14000156C
sub_140001624
sub_140001634
sub_140001650
start
__raise_securityfailure
__report_gsfailure
capture_previous_context
sub_14000195C
sub_140001998
sub_1400019D4
sub_140001A20
sub_140001AF8
sub_140001B94
sub_140001BB8
sub_140001BE4
sub_140001C34
sub_140001C4C
sub_140001CF8
sub_140001CFc
sub_140001D04
sub_140001D0C
Line 10 of 99
Output window
The initial autoanalysis has been finished.
AU: idle Down Disk: 75GB
Windows Type here to search E File Explorer Mail Task View Start 9:49 AM 4/11/2019
```

Assembly code for `sub_140001000`:

```
.idata:0000000140003000 ; =====
.idata:0000000140003000 ;
.idata:0000000140003000 ; Segment type: Externs
.idata:0000000140003000 ; _ida
.idata:0000000140003000 ; _stdcall CreateToolhelp32Snapshot(DWORD dwFlags, DWORD th32ProcessID)
.idata:0000000140003000     extrn CreateToolhelp32Snapshot:qword
.idata:0000000140003000 ; CODE XREF: sub_140001000+56tp
.idata:0000000140003000 ; DATA XREF: sub_140001000+56tr ...
.idata:0000000140003008 ; BOOL _stdcall Process32FirstW(HANDLE hSnapshot, LPPROCESSENTRY32W lppe)
.idata:0000000140003008 ; DATA XREF: sub_140001000+56tr ...
.idata:0000000140003028 ; _stdcall VirtualAllocEx(HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize, DWORD dwAllocationType, DWORD dwProtect)
.idata:0000000140003028     extrn VirtualAllocEx:qword
.idata:0000000140003028 ; CODE XREF: sub_140001000+166tp
.idata:0000000140003028 ; DATA XREF: sub_140001000+166tr
.idata:0000000140003030 ; _stdcall WriteProcessMemory(HANDLE hProcess, LPVOID lpBaseAddress, LPCVOID lpBuffer, SIZE_T nSize, SIZE_T *lpNumberOfBytesWritten)
.idata:0000000140003030     extrn WriteProcessMemory:qword
.idata:0000000140003030 ; CODE XREF: sub_140001000+1F2tp
.idata:0000000140003030 ; DATA XREF: sub_140001000+1F2tr
.idata:0000000140003038 ; _stdcall VirtualAllocEx(HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize, DWORD dwAllocationType, DWORD dwProtect)
.idata:0000000140003038     extrn VirtualAllocEx:qword
.idata:0000000140003038 ; CODE XREF: sub_140001000+166tp
.idata:0000000140003038 ; DATA XREF: sub_140001000+166tr
00001A00 0000000140003000: .idata:CreateToolhelp32Snapshot (Synchronized with Hex View-1)
```

Let's click the first spot and be greeted with the assembly block of where CreateToolhelp32Snapshot is in.

At this point let's start diving into dynamic analysis.

The screenshot shows the IDA Pro interface with the assembly window active. The assembly code for the `__security_check_cookie` function is displayed:

```
push    rdi
sub    rsp, 2A0h
mov    rax, cs:_security_cookie
xor    rax, rsp
mov    [rsp+2A8h+var_18], rax
mov    rcx, cs:?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::basic_ostream<char, std::char_traits<char>> std::cout
lea    rdx, aTest      ; "Test!\n"
call   sub_140001320
xor    edx, edx       ; th32ProcessID
mov    [rsp+2A8h+pe.dwSize], 238h
mov    rax, 64617065746F6Eh
mov    qword ptr [rsp+2A8h+SubStr], rax
lea    ecx, [rdx+2]    ; dwFlags
call   cs:CreateToolhelp32Snapshot
mov    rsi, rax
cmp    rax, 0xFFFFFFFFFFFFFFFh
jnz    short loc_140001074
```

The assembly window has several annotations:

- A red box highlights the instruction `call cs:CreateToolhelp32Snapshot`.
- A green box highlights the instruction `mov rsi, rax`.
- A yellow box highlights the instruction `cmp rax, 0xFFFFFFFFFFFFFFFh`.
- Callouts show the assembly for `loc_140001074`, `loc_1400010A2`, and `sub_140001000+56`.
- The status bar at the bottom indicates: `100.00% (180,806) (349,303) 00000456 0000000140001056: sub_140001000+56 (Synchronized with Hex View-1)`.
- The bottom taskbar shows the Windows Start button, a search bar, and various pinned icons.

Exports

- We'll discuss exports in a later example, for now know that exports are pretty important for getting a starting point in some cases.

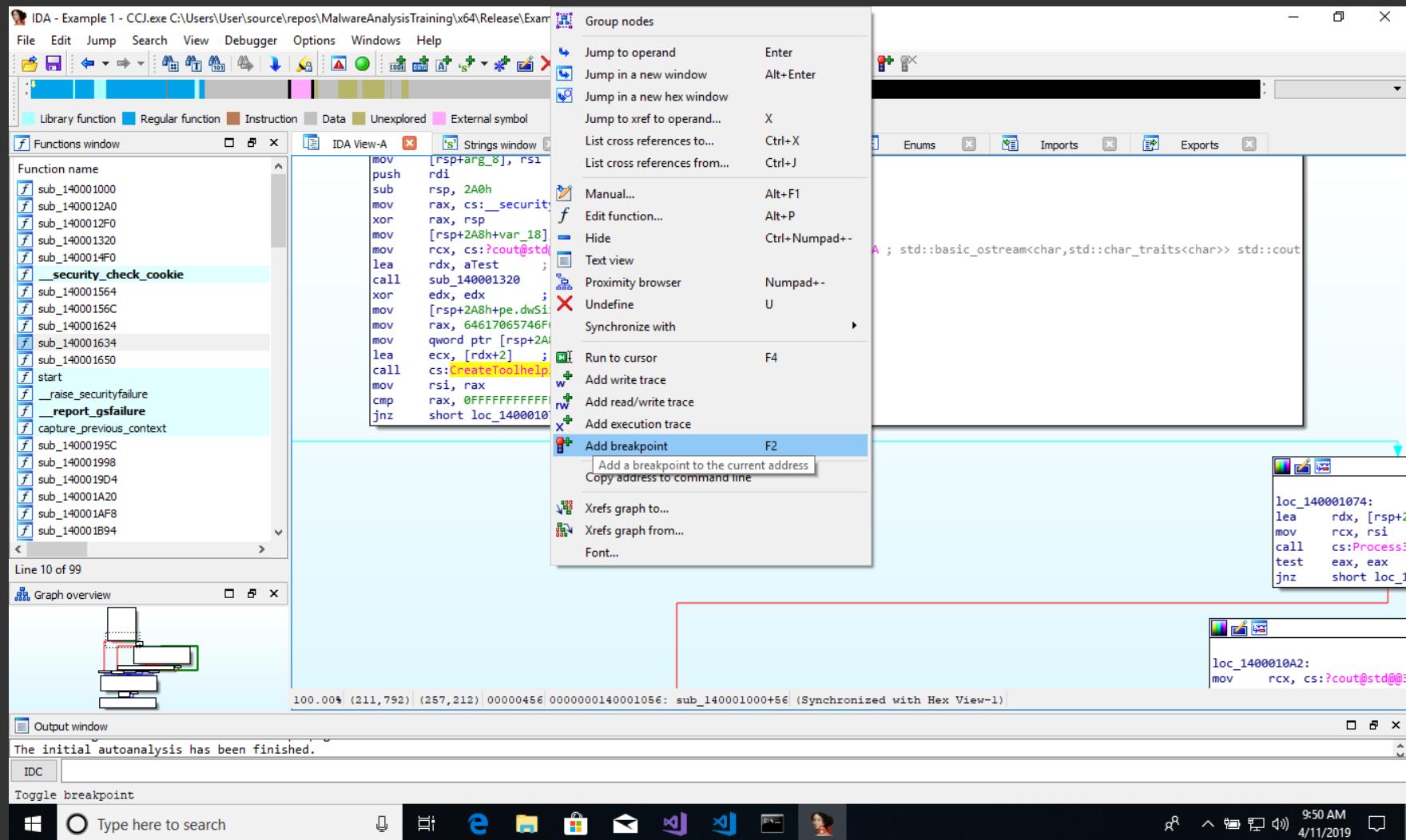
Dynamic Analysis

- Allows one to inspect data, memory, and functions at runtime.
- Using breakpoints and debugging the malware we can come up with faster assumptions while we run it.

When to use Dynamic Analysis?

- When the malware/binary becomes too difficult to figure out from static analysis alone.
- Assembly takes quite some time to drudge through and dynamic analysis can quickly find answers that static analysis may not have immediately.
- The only negative is that you have to run the malware (Better hope there's no VM escape in the malware)

Let's create a breakpoint on "CreateToolhelp32Snapshot". Do this by clicking the line that the pink text of the function is on and right click and hover down to the "Add breakpoint" option.



Go back to the “Imports” tab and do this for a few other functions like

WriteProcessMemory

CreateRemoteThread

The screenshot shows the IDA Pro interface with the assembly view open. The assembly window displays the following code:

```
loc_14000119A:
mov    rcx, cs:?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::basic_ostream<char, std::char_traits<char>> std::cout
call   sub_140001320
lea    rdx, sub_1400014F0
mov    rcx, rax
call   cs:??6?$basic_ostream@DU?$char_traits@D@std@@@std@@QEAEEAV01@P6AAEAV01@AEAV01@@Z@Z ; std::basic_ostream<char, std::char_traits<char>>::operator<<
xor   edx, edx      ; lpAddress
mov    [rsp+2A8h+f1Protect], 40h ; f1Protect
mov    r9d, 3000h      ; flAllocationType
mov    r8d, 102h       ; dwSize
mov    rcx, rbx       ; hProcess
call   cs:VirtualAllocEx
xor   esi, esi
lea    r8, unk_140005040 ; lpBuffer
mov    rdx, rax       ; lpBaseAddress
mov    qword ptr [rsp+2A8h+f1Protect], rsi ; lpNumberOfBytesWritten
mov    r9d, 102h       ; nSize
mov    rcx, rbx       ; hProcess
mov    rdi, rax
call   cs:WriteProcessMemory
mov    [rsp+2A8h+lpThreadId], rsi ; lpThreadId
mov    r9, rdi         ; lpStartAddress
mov    [rsp+2A8h+dwCreationFlags], esi ; dwCreationFlags
xor   r8d, r8d        ; dwStackSize
xor   edx, edx        ; lpThreadAttributes
mov    qword ptr [rsp+2A8h+f1Protect], rsi ; lpParameter
mov    rcx, rbx       ; hProcess
call   cs>CreateRemoteThread
mov    rdi, rax
test  rax, rax
jz    short loc_140001248
```

The assembly code is annotated with comments explaining the parameters and purpose of each instruction. The `WriteProcessMemory` call is highlighted with a red rectangle. The `CreateRemoteThread` call is also visible in the code.

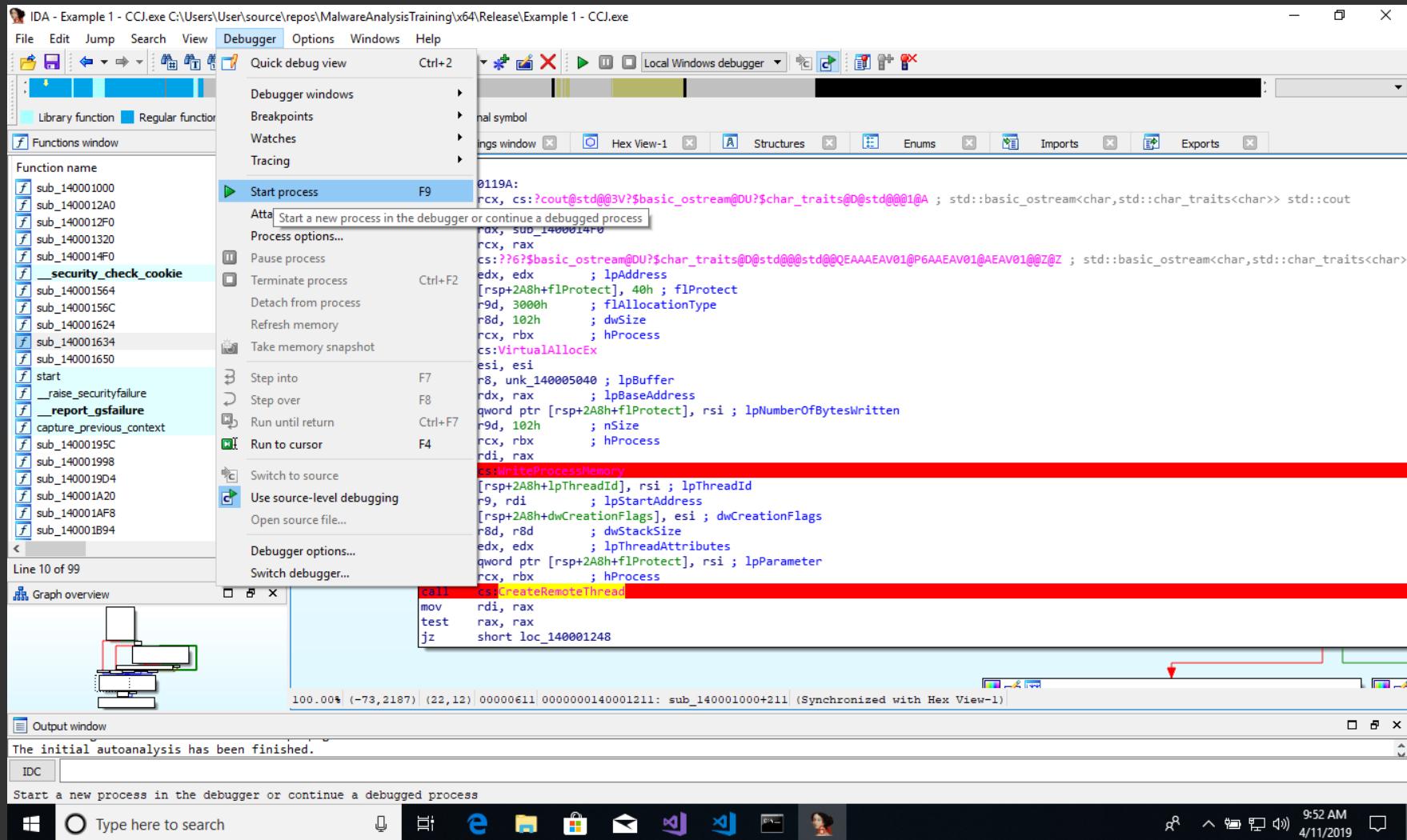
These are some pretty important functions. So just trust me on breakpointing them for now.

The screenshot shows the IDA Pro interface with the assembly view open. The assembly window displays the following code:

```
loc_14000119A:
mov    rcx, cs:@cout@0@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::basic_ostream<char, std::char_traits<char>> std::cout
call   sub_140001320
lea    rdx, sub_1400014F0
mov    rcx, rax
call   cs:@?6?$basic_ostream@DU?$char_traits@D@std@@@std@@QEAEEAV01@P6AAEAV01@AEAV01@0Z@Z ; std::basic_ostream<char, std::char_traits<char>>
xor    edx, edx      ; lpAddress
mov    [rsp+2A8h+f1Protect], 40h ; f1Protect
mov    r9d, 3000h      ; flAllocationType
mov    r8d, 102h        ; dwSize
mov    rcx, rbx        ; hProcess
call   cs:VirtualAllocEx
xor    esi, esi
lea    r8, unk_140005040 ; lpBuffer
mov    rdx, rax        ; lpBaseAddress
mov    qword ptr [rsp+2A8h+f1Protect], rsi ; lpNumberOfBytesWritten
mov    r9d, 102h        ; nSize
mov    rcx, rbx        ; hProcess
mov    rdi, rax
call   cs:WriteProcessMemory
mov    [rsp+2A8h+lpThreadId], rsi ; lpThreadId
mov    r9, rdi          ; lpStartAddress
mov    [rsp+2A8h+dwCreationFlags], esi ; dwCreationFlags
xor    r8d, r8d          ; dwStackSize
xor    edx, edx          ; lpThreadAttributes
mov    qword ptr [rsp+2A8h+f1Protect], rsi ; lpParameter
mov    rcx, rbx        ; hProcess
call   cs>CreateRemoteThread
mov    rdi, rax
test   rax, rax
jz    short loc_140001248
```

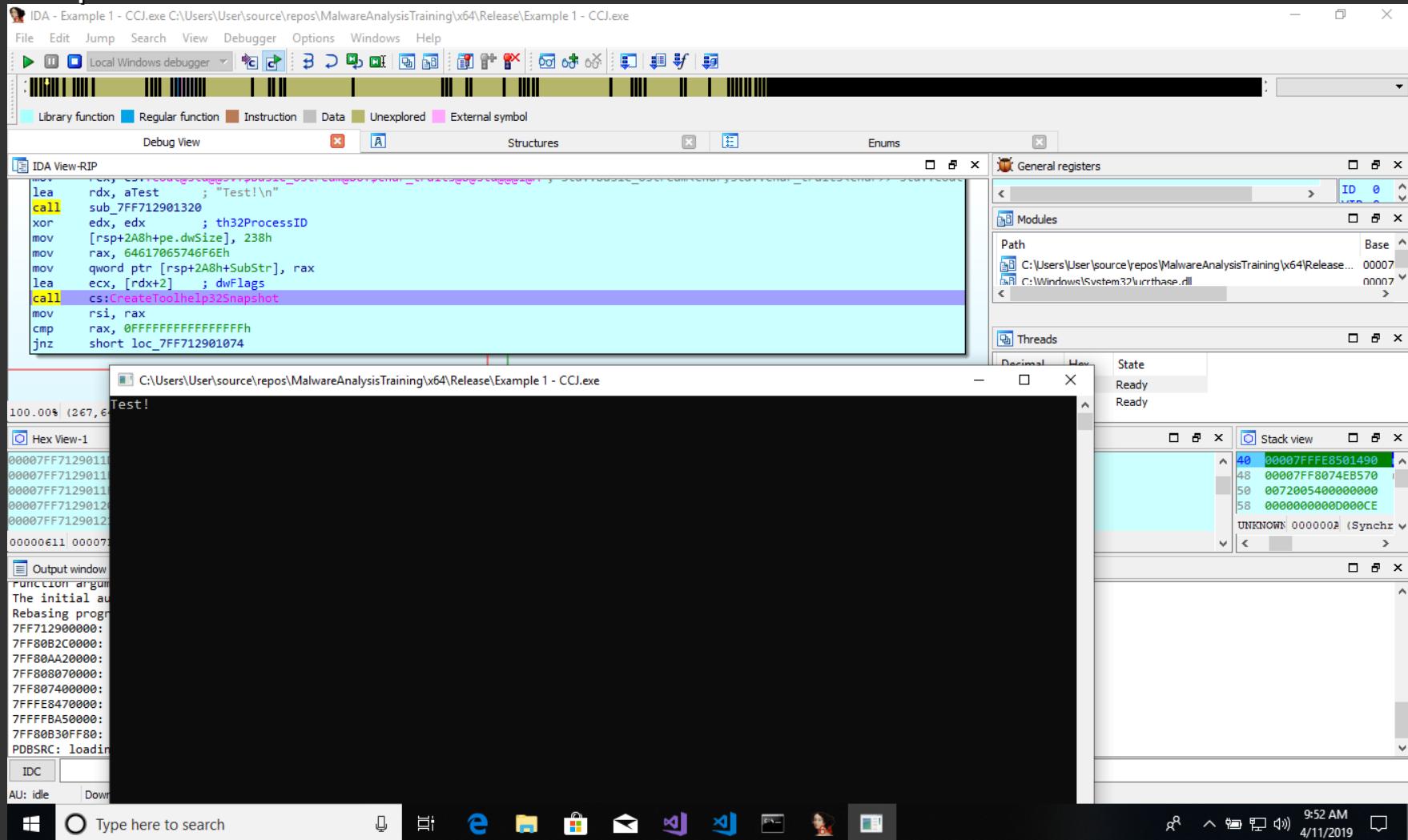
The assembly code is annotated with several red highlights, specifically on the calls to `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread`. The output window at the bottom left of the IDA interface shows the message "The initial autoanalysis has been finished."

Let's go ahead and now start up the malware again in IDA.

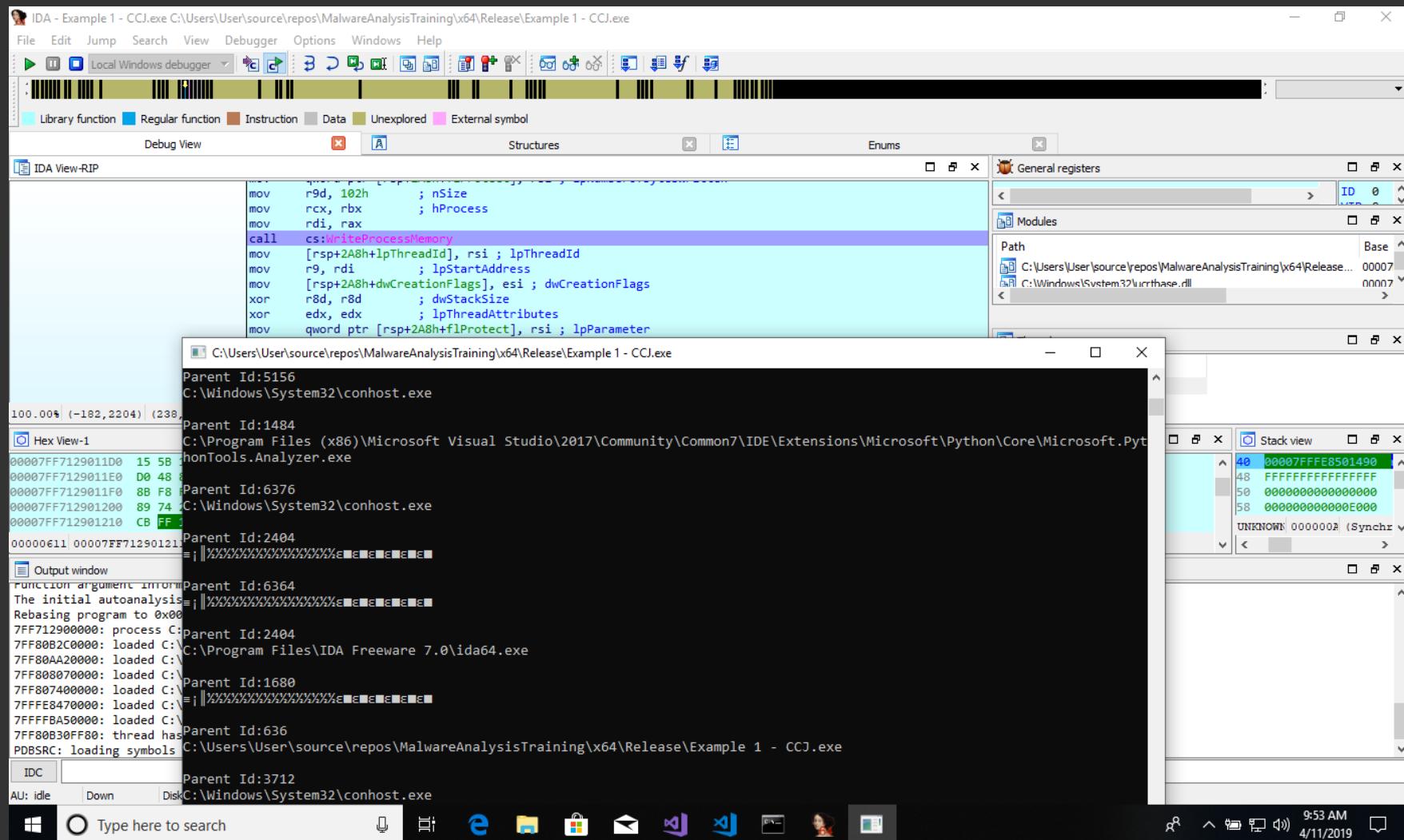


We notice that we stopped at CreateToolhelp32Snapshot because it is now colored purple.

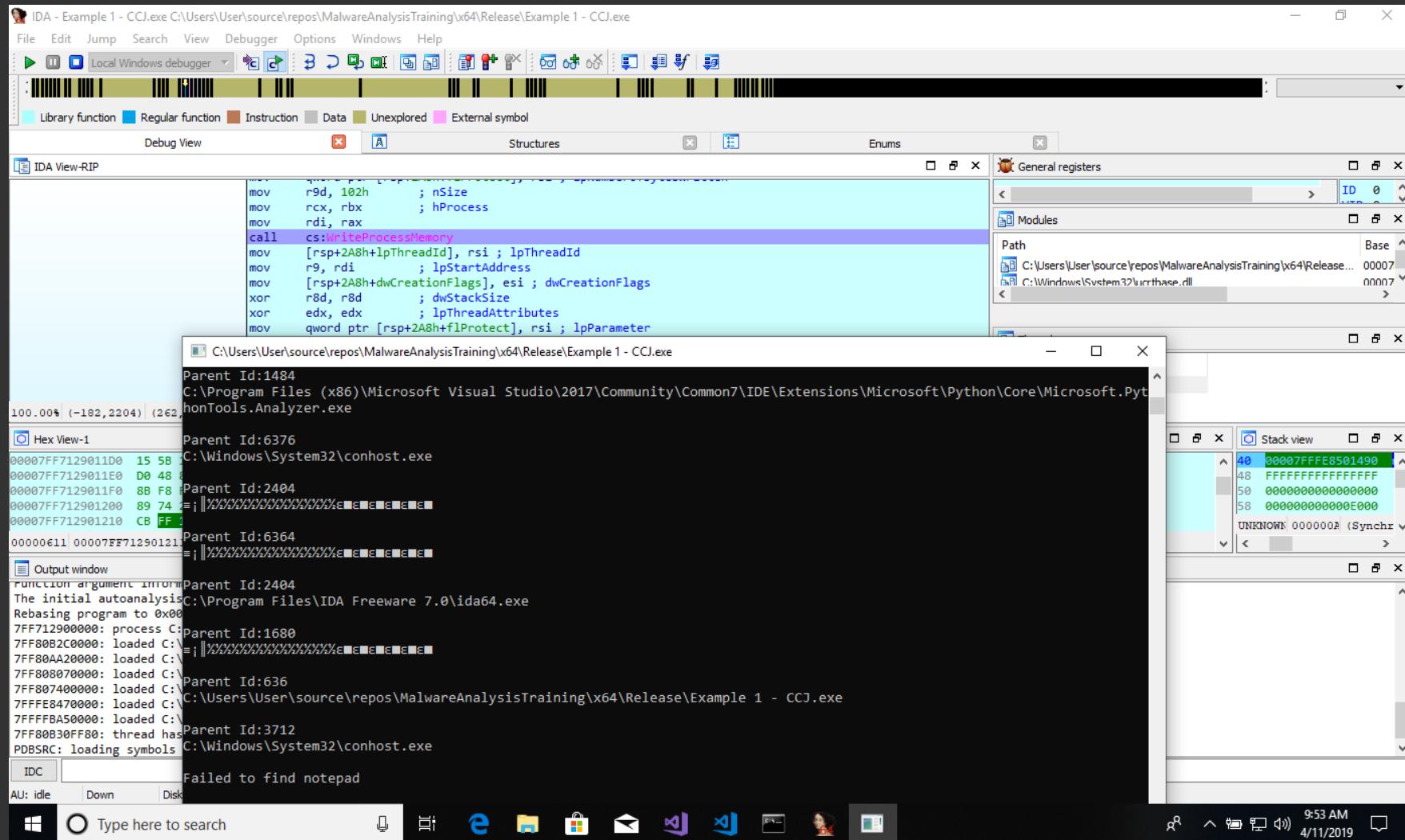
Pulling the command prompt up for the malware shows us it has only printed "Test!" so far. This means the process output doesn't happen till after the snapshot.



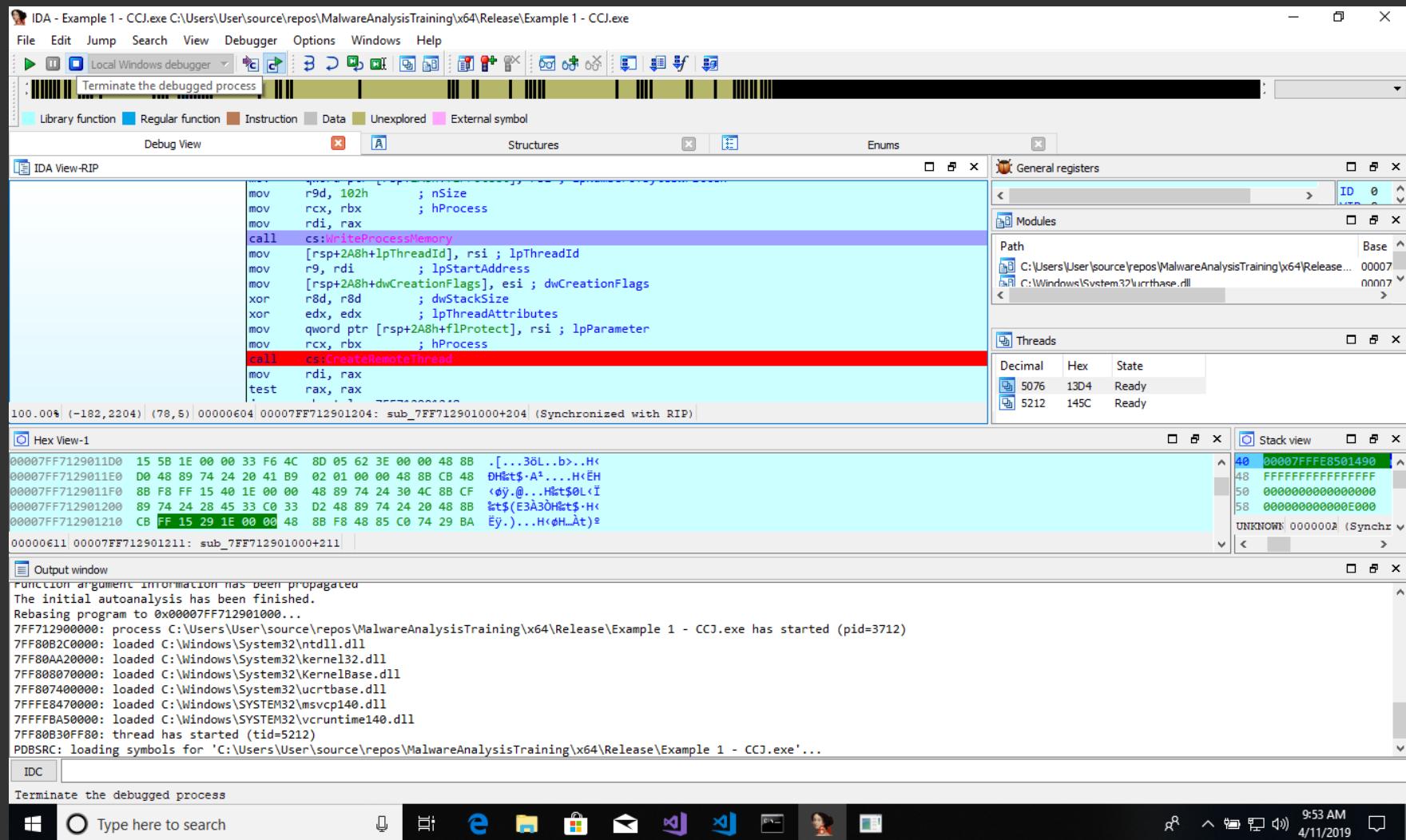
Hitting the play button in the top left. The next stopping point should be "WriteProcessMemory". Now we see all the output of the processes again.



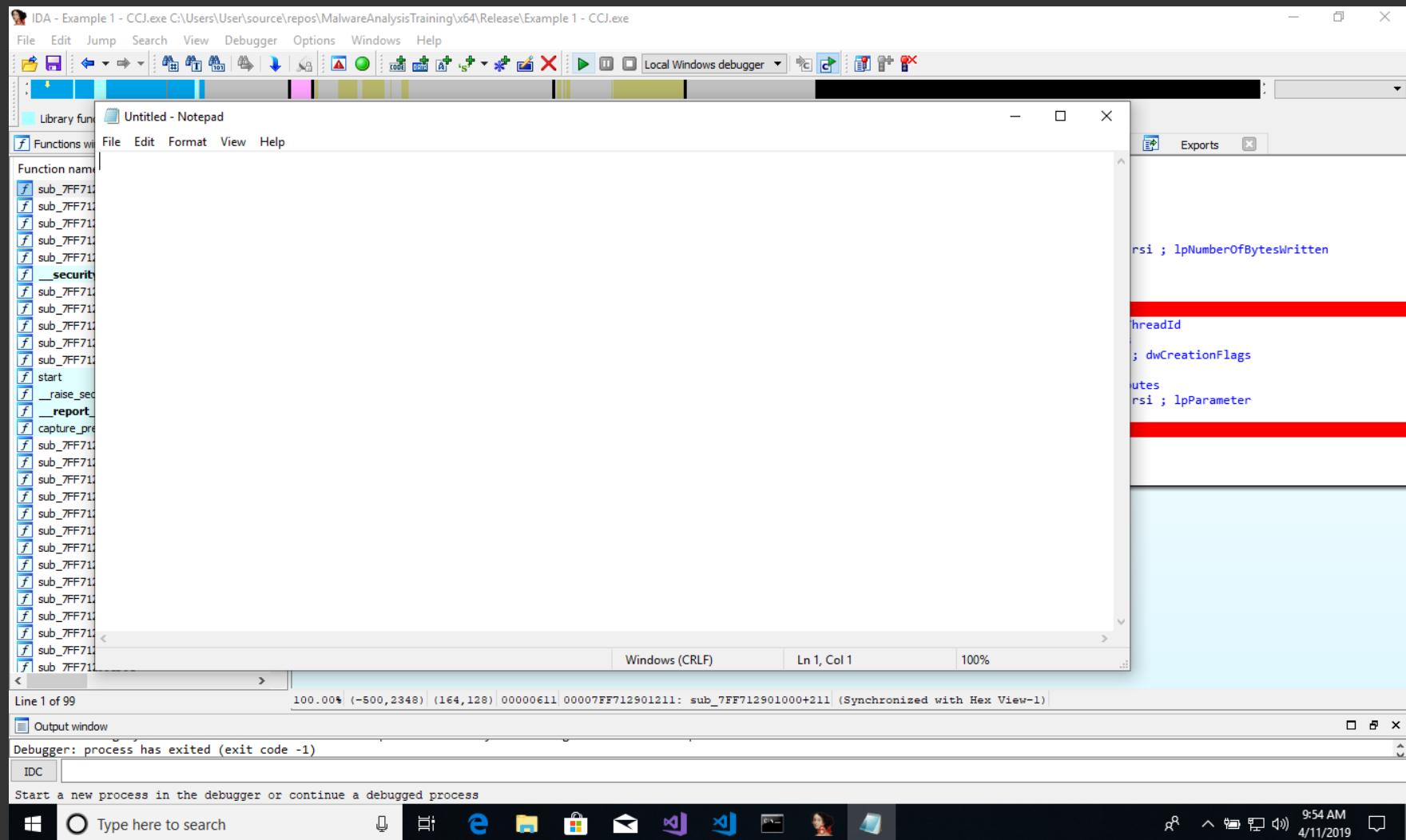
We'll scroll down the prompt a bit and see "Failed to find notepad" again. So long story short we've confirmed in the timeline that the Snapshot is used for listing the processes and the hypothesis now is that it must be looking for a notepad process.



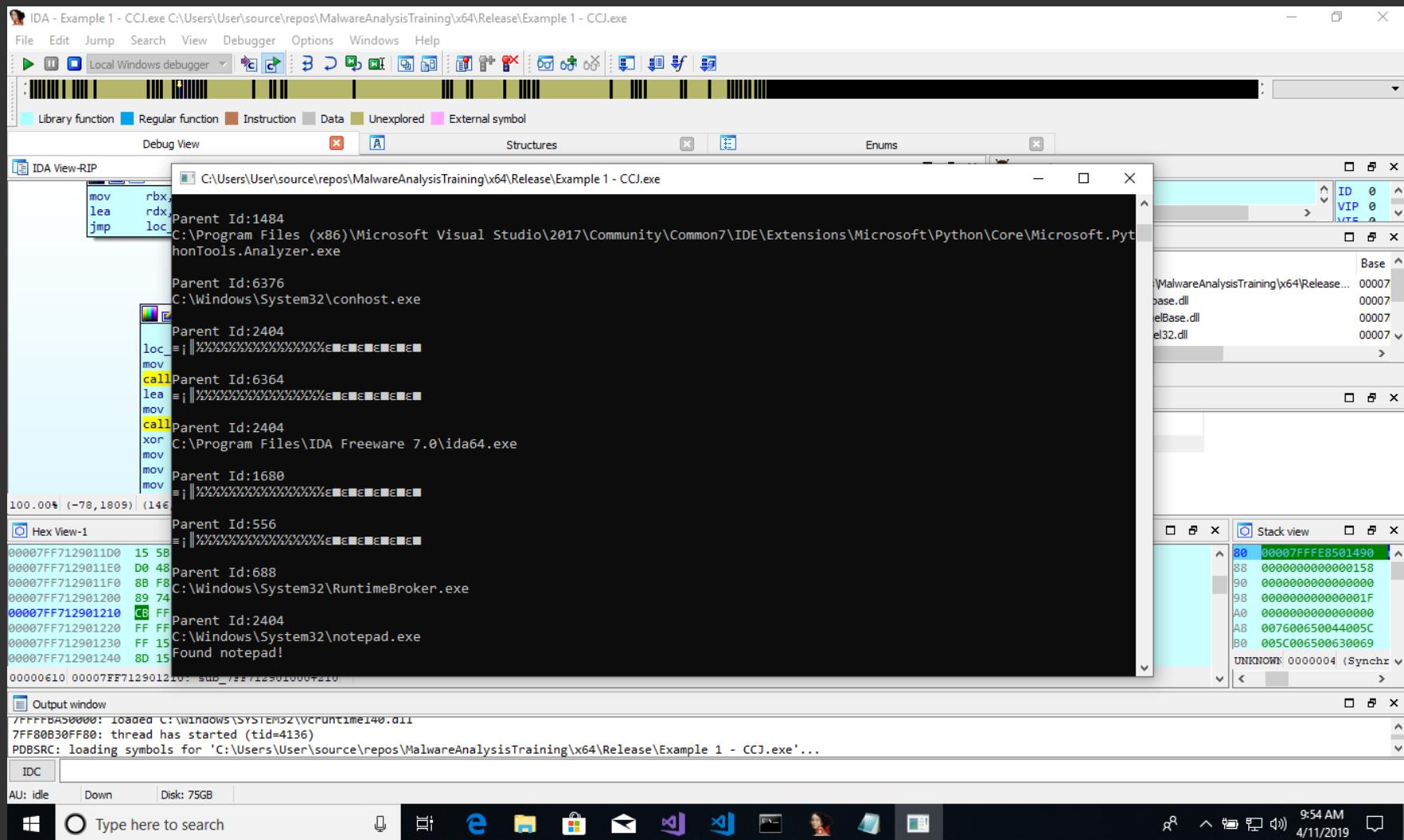
Lets go ahead and restart the malware in IDA.



This time lets open a notepad and watch what happens to confirm a few things.



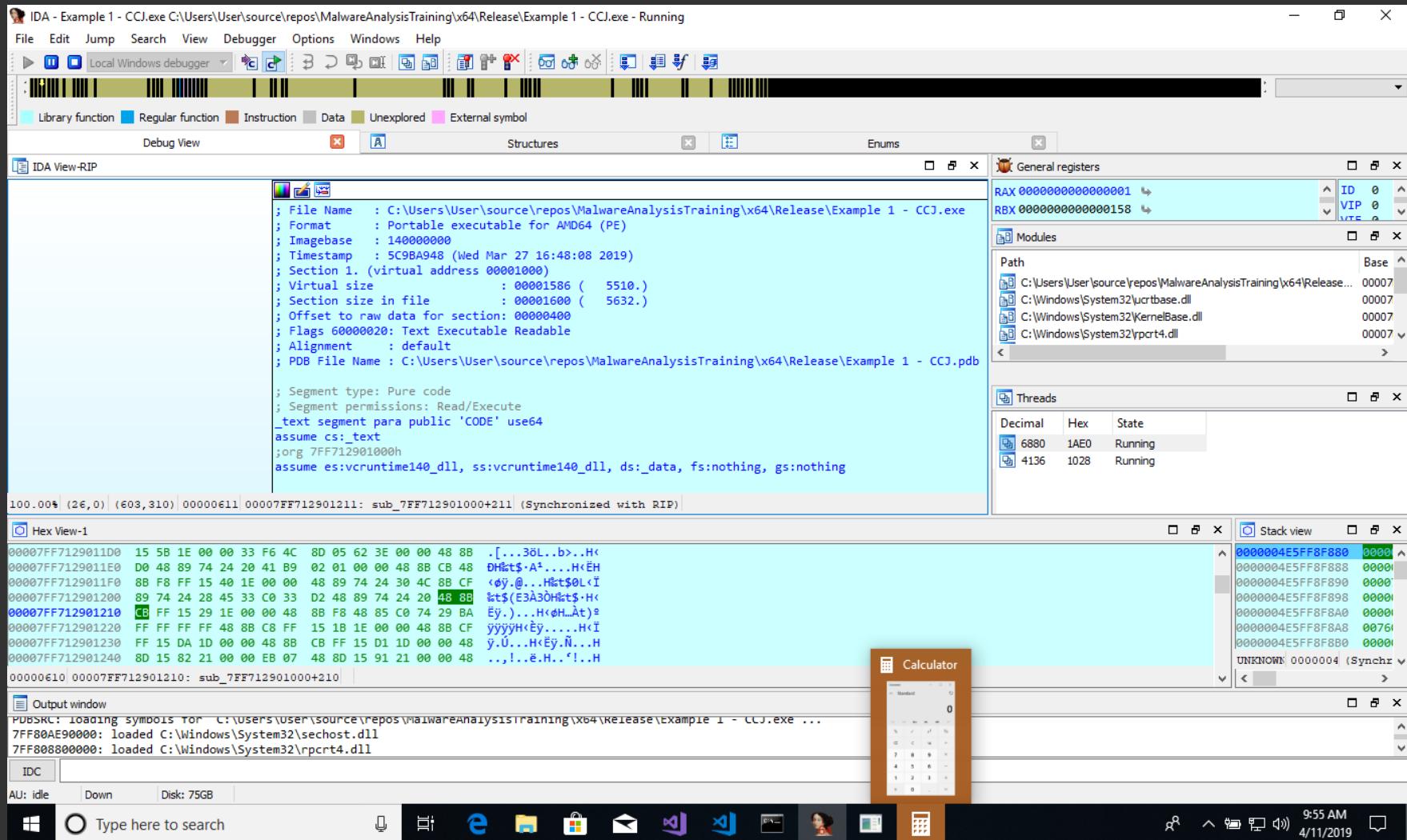
Running it again until we hit “WriteProcessMemory” shows us the “Found notepad!” message. So our hypothesis is shown true. It is looking for a notepad.



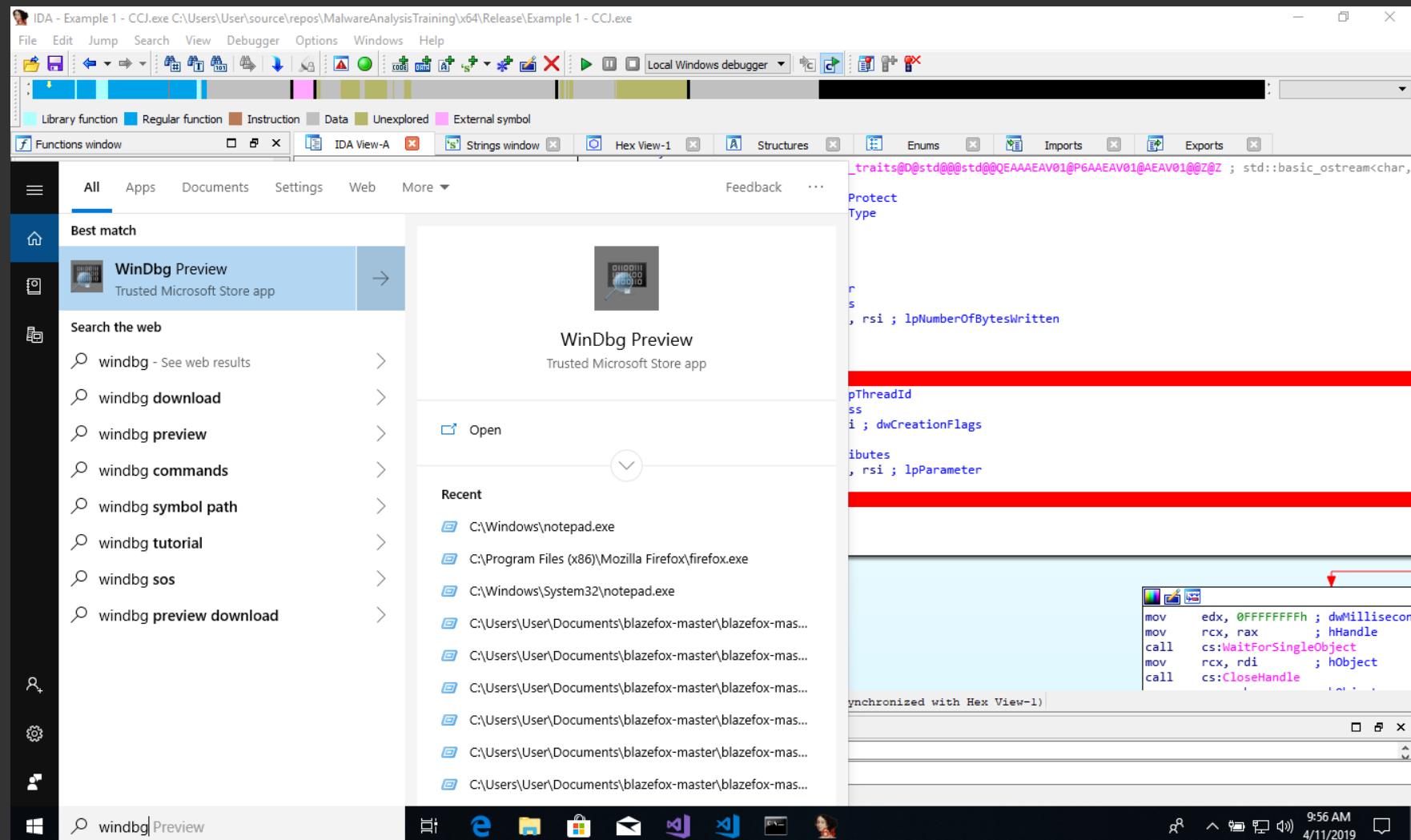
Now hit play until no more breakpoints are hit and watch what happens.

The calculator pops up, but we notice that the notepad vanishes along with it.

Time to get to the bottom of this.

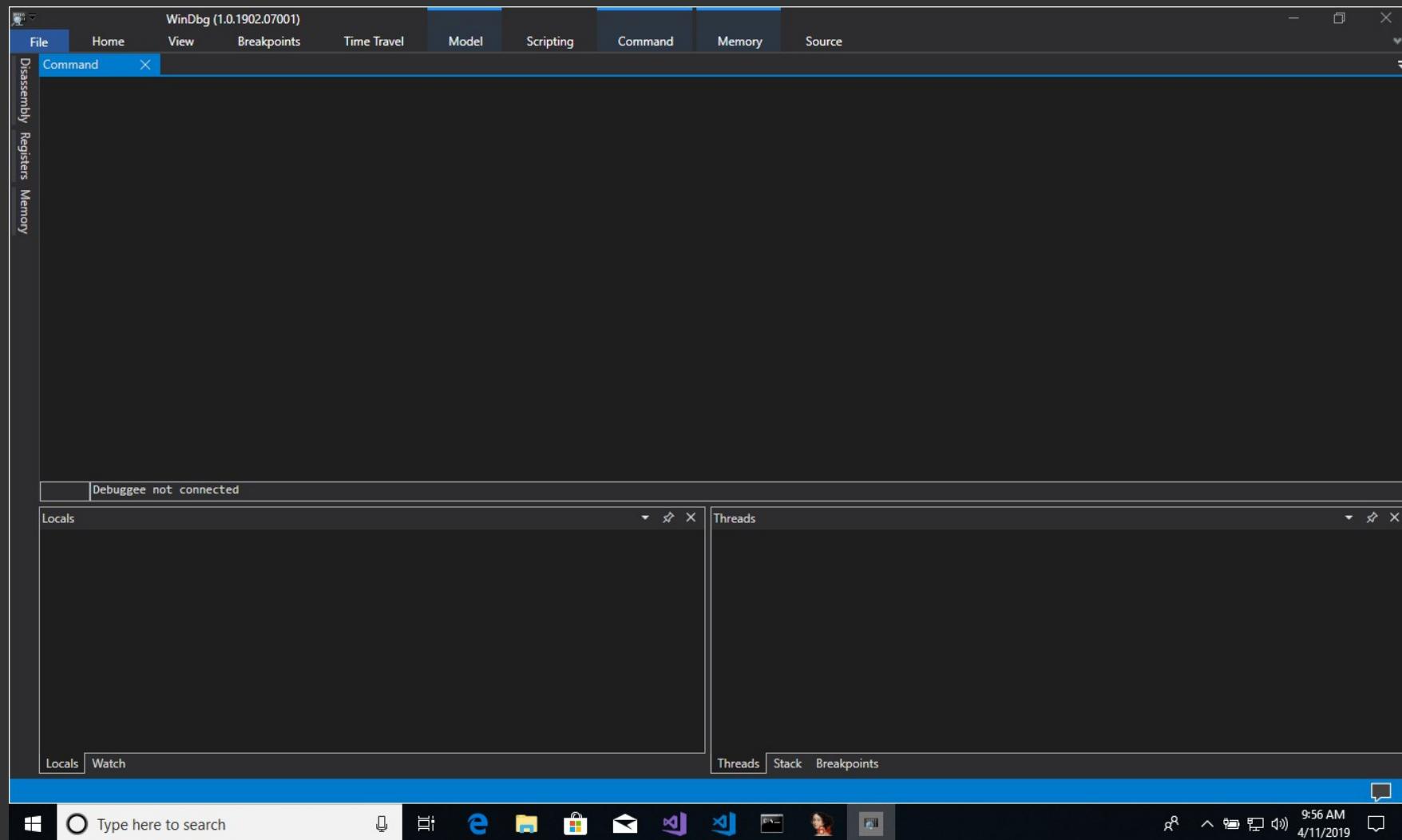


Time to open windbg.

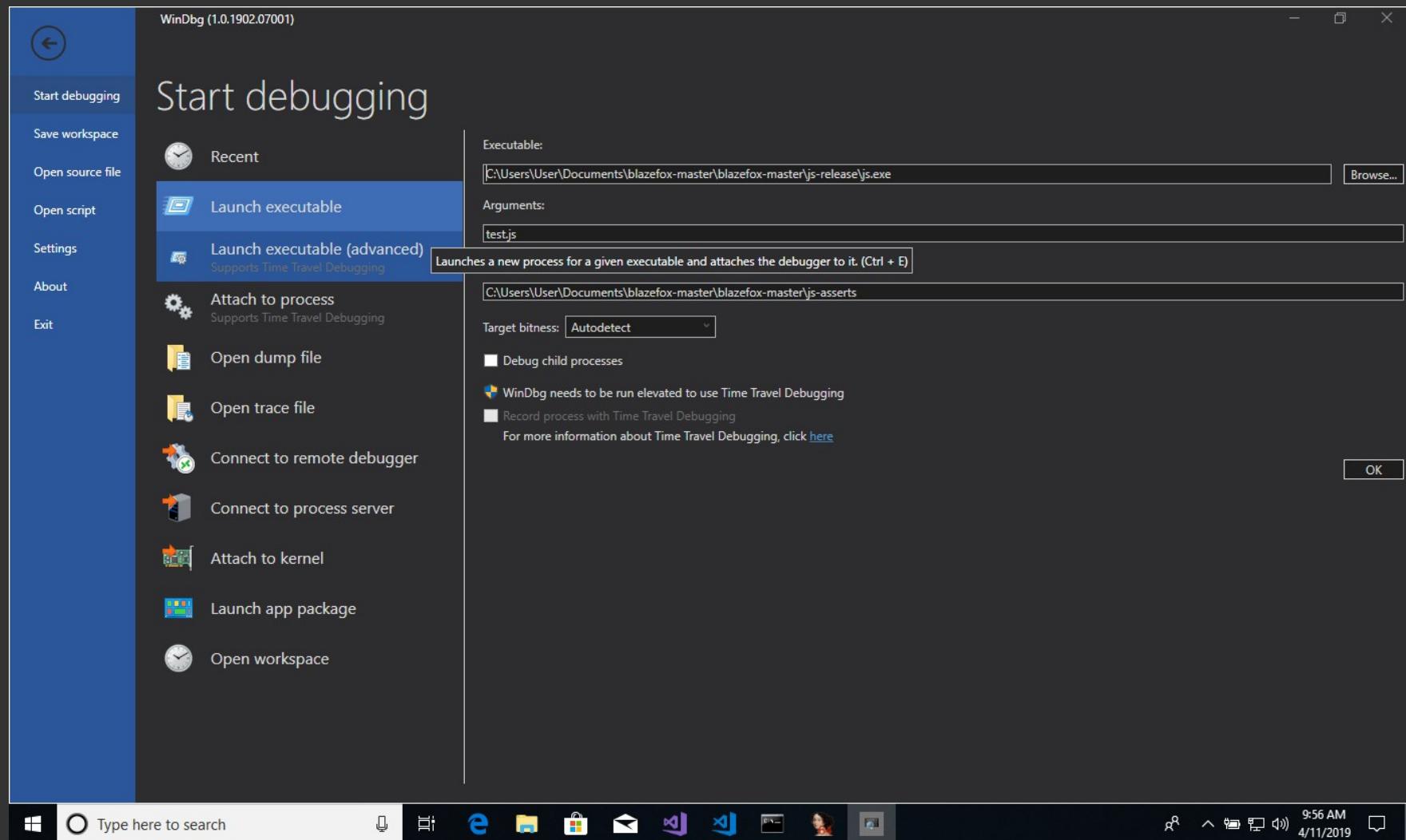


Once windbg is opened it should look like this in some way.

I'm currently using the dark skin variant.

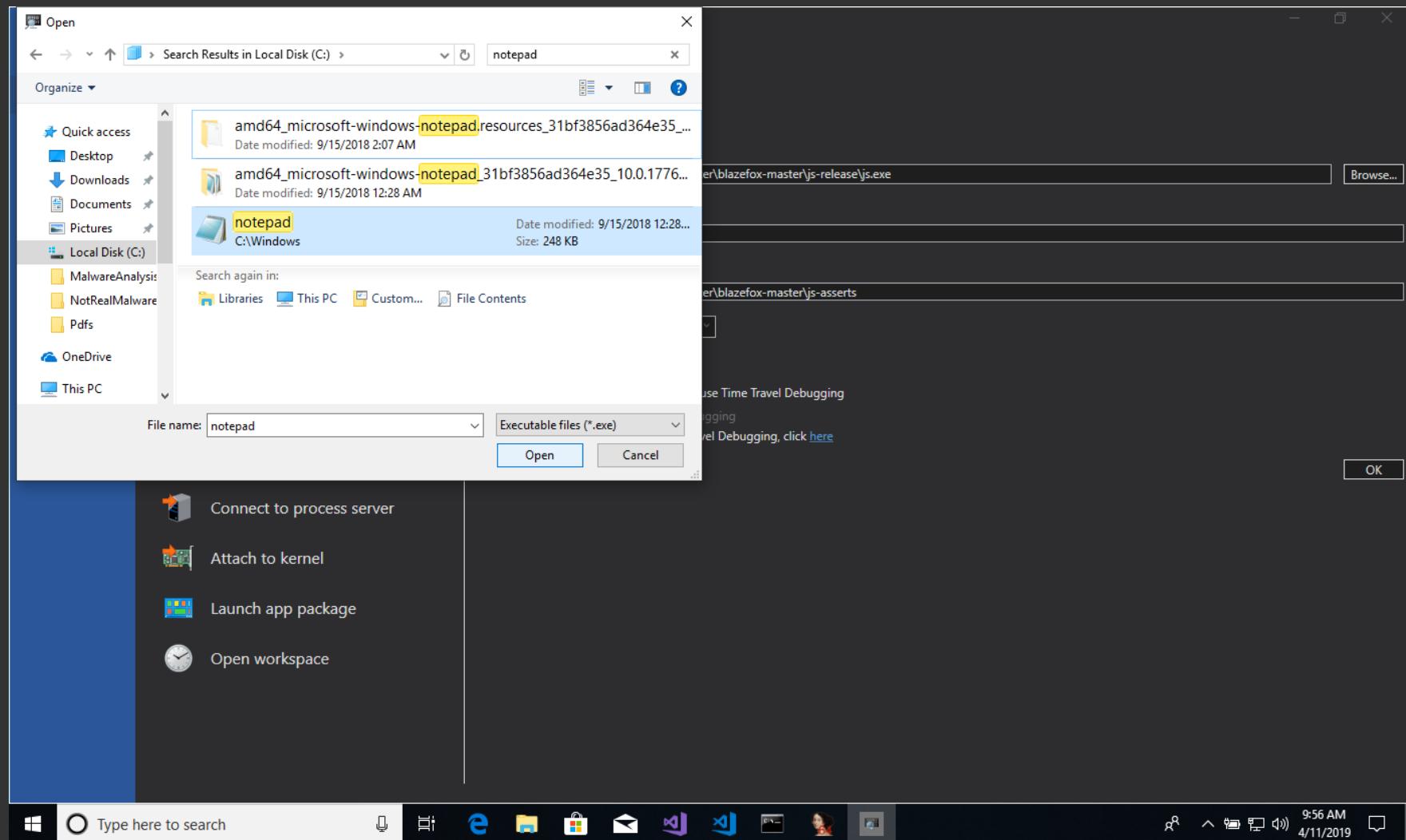


Click File in the top left and let's click “Launch executable”.



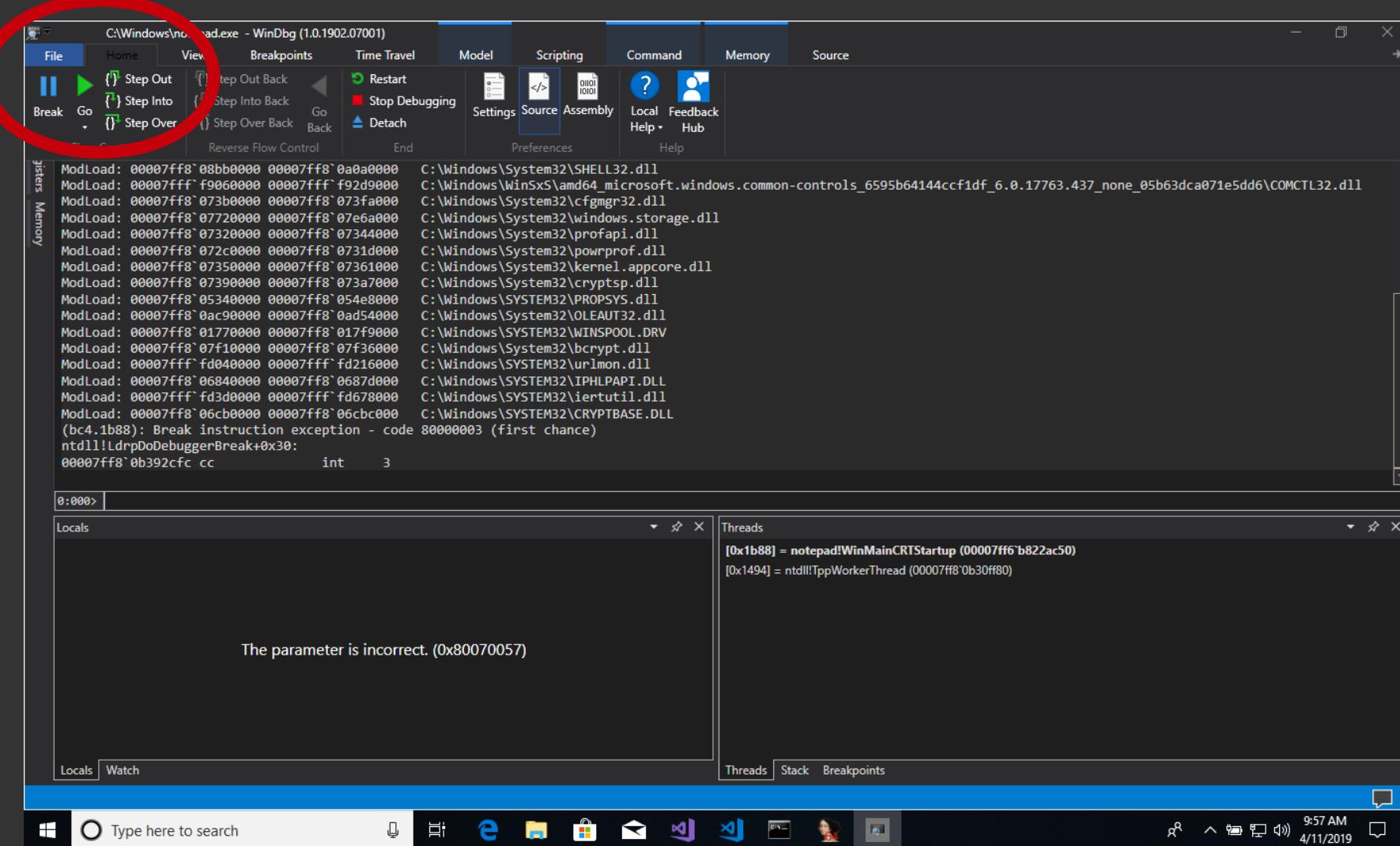
Open notepad.

Notepad tends to reside in system32, but if you click the Local Disk C: and type notepad in the search, notepad should pop up pretty fast.

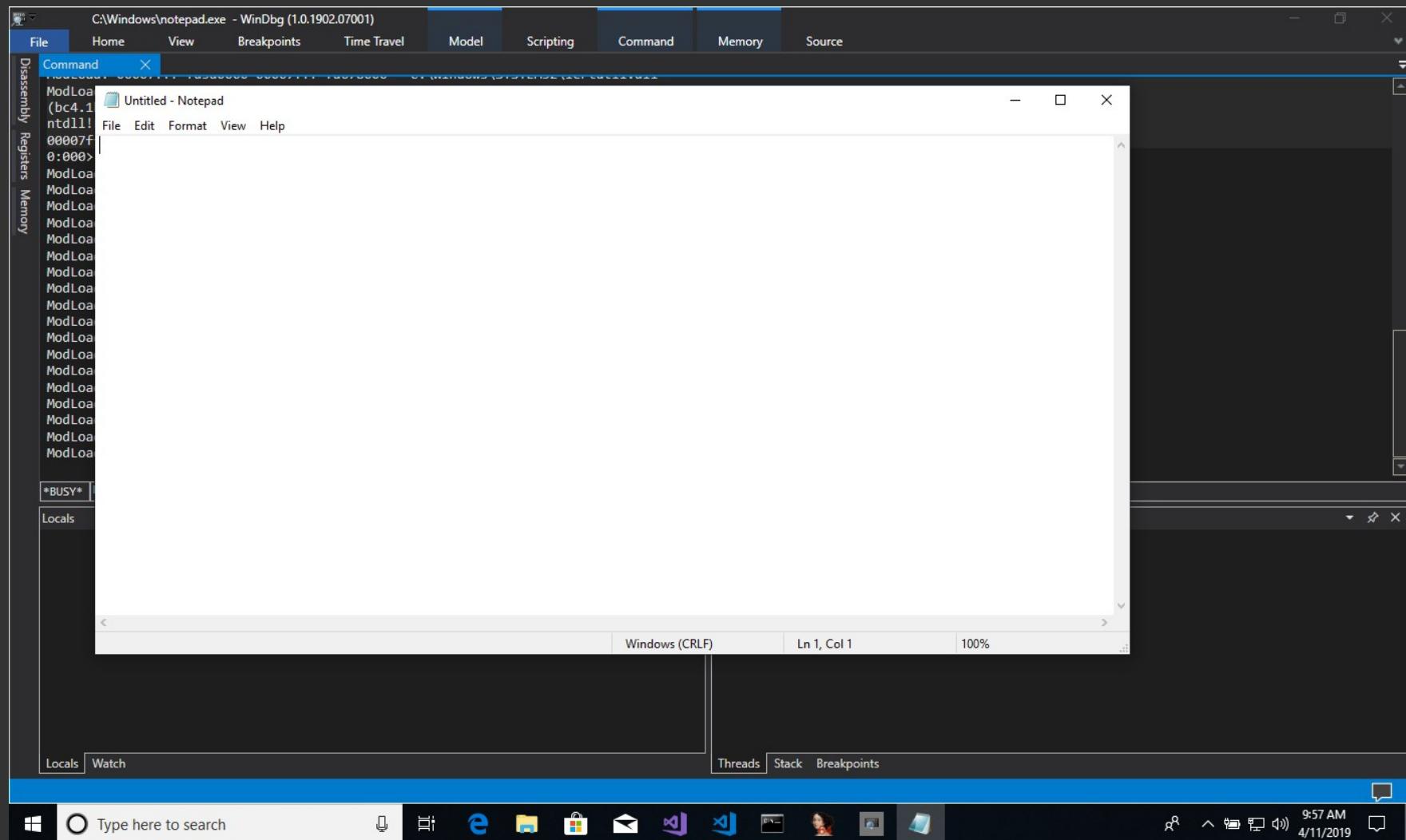


After hitting open, we should see a screen like this and windbg should now be watching notepad.

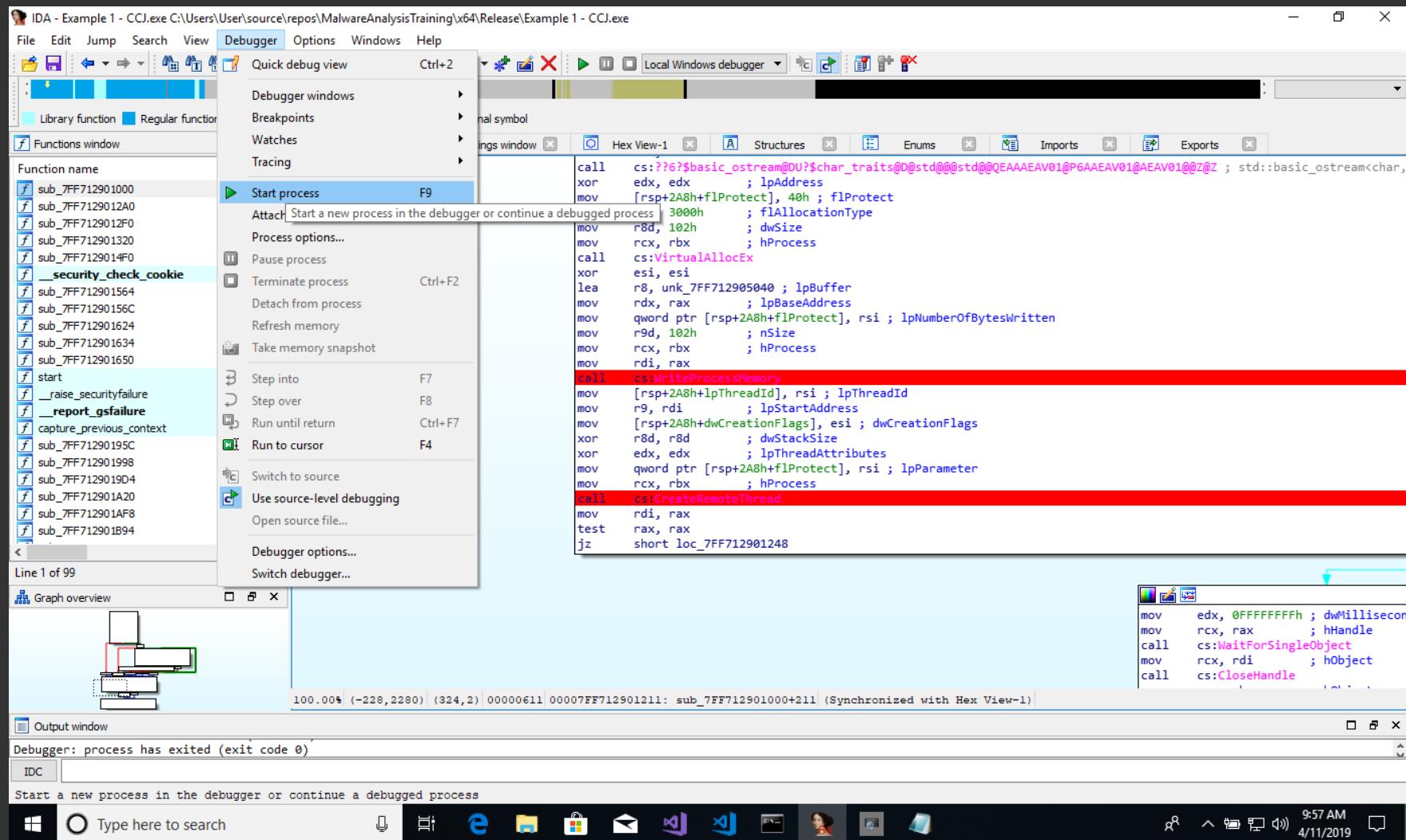
Hitting the giant “Go” play button to start notepad.



We should now see our notepad start running and windbg running in the background.

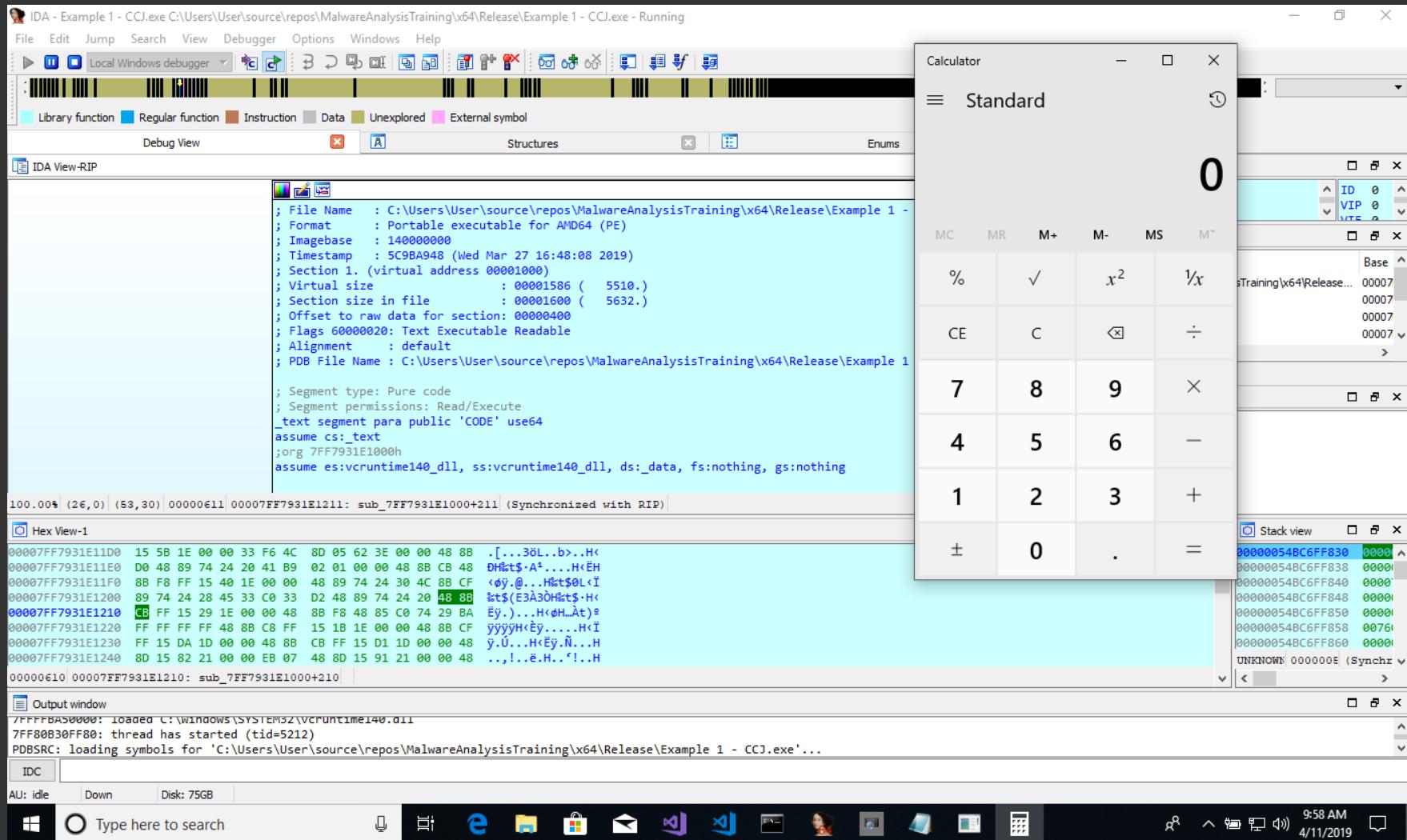


Head back into IDA and let us start the malware process over again and watch what happens.



Hit the play button a few times to get through all the breakpoints.

We should now see calculator spawn and windbg should now light up about a few things that happened.



Opening up windbg shows us an Access violation has occurred and the notepad crashed.

This at least tells us why the notepad vanishes shortly before the calculator spawns.

```
C:\Windows\notepad.exe - WinDbg (1.0.1902.07001)
File Home View Breakpoints Time Travel Model Scripting Command Memory Source
Command X
0:000> g
ModLoad: 00007ff8`086c0000 00007fff`086ee000 C:\Windows\System32\IMM32.DLL
ModLoad: 00007ff8`058c0000 00007fff`0595c000 C:\Windows\system32\uxtheme.dll
ModLoad: 00007ff8`086f0000 00007fff`08792000 C:\Windows\System32\clbcatq.dll
ModLoad: 00007fff`fea40000 00007fff`feb47000 C:\Windows\System32\MMCoreR.dll
ModLoad: 00007ff8`0a8b0000 00007fff`0aa1a000 C:\Windows\System32\MSCTF.dll
ModLoad: 00007ff8`05cd0000 00007fff`05cfe000 C:\Windows\system32\dwmapi.dll
ModLoad: 00007ff8`08310000 00007fff`084eb000 C:\Windows\System32\CRYPT32.dll
ModLoad: 00007ff8`07370000 00007fff`07382000 C:\Windows\System32\MSASN1.dll
ModLoad: 00007fff`e4200000 00007fff`e42c6000 C:\Windows\System32\efswrtr.dll
ModLoad: 00007fff`f5810000 00007fff`f582b000 C:\Windows\System32\MPR.dll
ModLoad: 00007ff8`035e0000 00007fff`03733000 C:\Windows\SYSTEM32\wintypes.dll
ModLoad: 00007ff8`059d0000 00007fff`05bdd000 C:\Windows\System32\twinapi.appcore.dll
ModLoad: 00007ff8`05980000 00007fff`059a8000 C:\Windows\System32\RMCLIENT.dll
ModLoad: 00007fff`f2a00000 00007fff`f2a6c000 C:\Windows\System32\oleacc.dll
ModLoad: 00007fff`fe860000 00007fff`fe8f5000 C:\Windows\System32\TextInputFramework.dll
ModLoad: 00007ff8`037b0000 00007fff`03ad2000 C:\Windows\System32\CoreUIComponents.dll
ModLoad: 00007ff8`04c00000 00007fff`04ce2000 C:\Windows\System32\CoreMessaging.dll
ModLoad: 00007ff8`063e0000 00007fff`06411000 C:\Windows\SYSTEM32\ntmarta.dll
(bc4.738): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
00000238`7f130055 0000 add byte ptr [rax],al ds:00000000`00000021=??
0:005>
```

The scope specified was not found. (0x8007013e)

Locals Watch

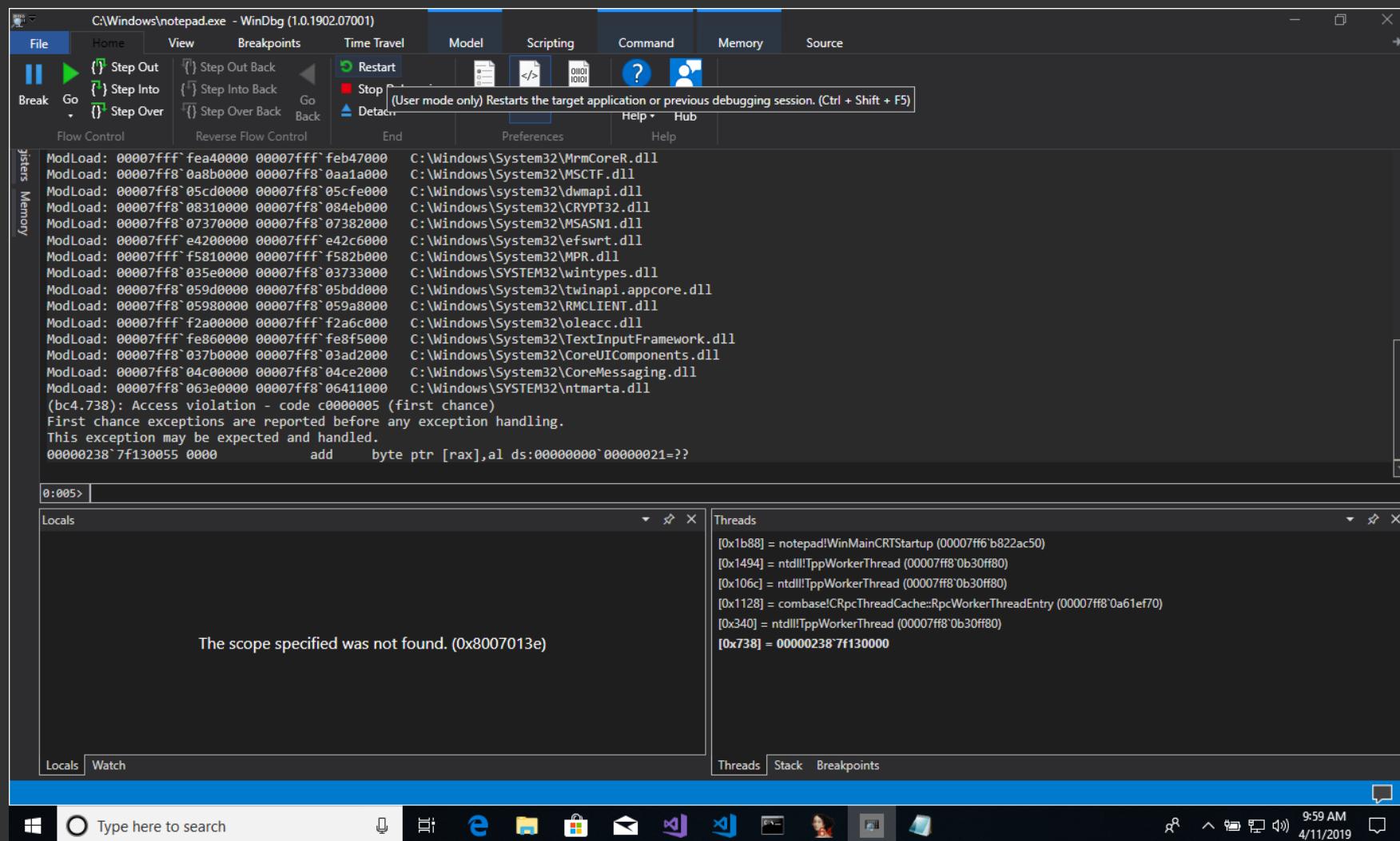
Threads

[0x1b88] = notepad!WinMainCRTStartup (00007ff6`b822ac50)
[0x1494] = ntdll!TppWorkerThread (00007ff8`0b30ff80)
[0x106c] = ntdll!TppWorkerThread (00007ff8`0b30ff80)
[0x1128] = combase!CRpcThreadCache::RpcWorkerThreadEntry (00007ff8`0a61ef70)
[0x340] = ntdll!TppWorkerThread (00007ff8`0b30ff80)
[0x738] = 00000238`7f130000

9:58 AM 4/11/2019

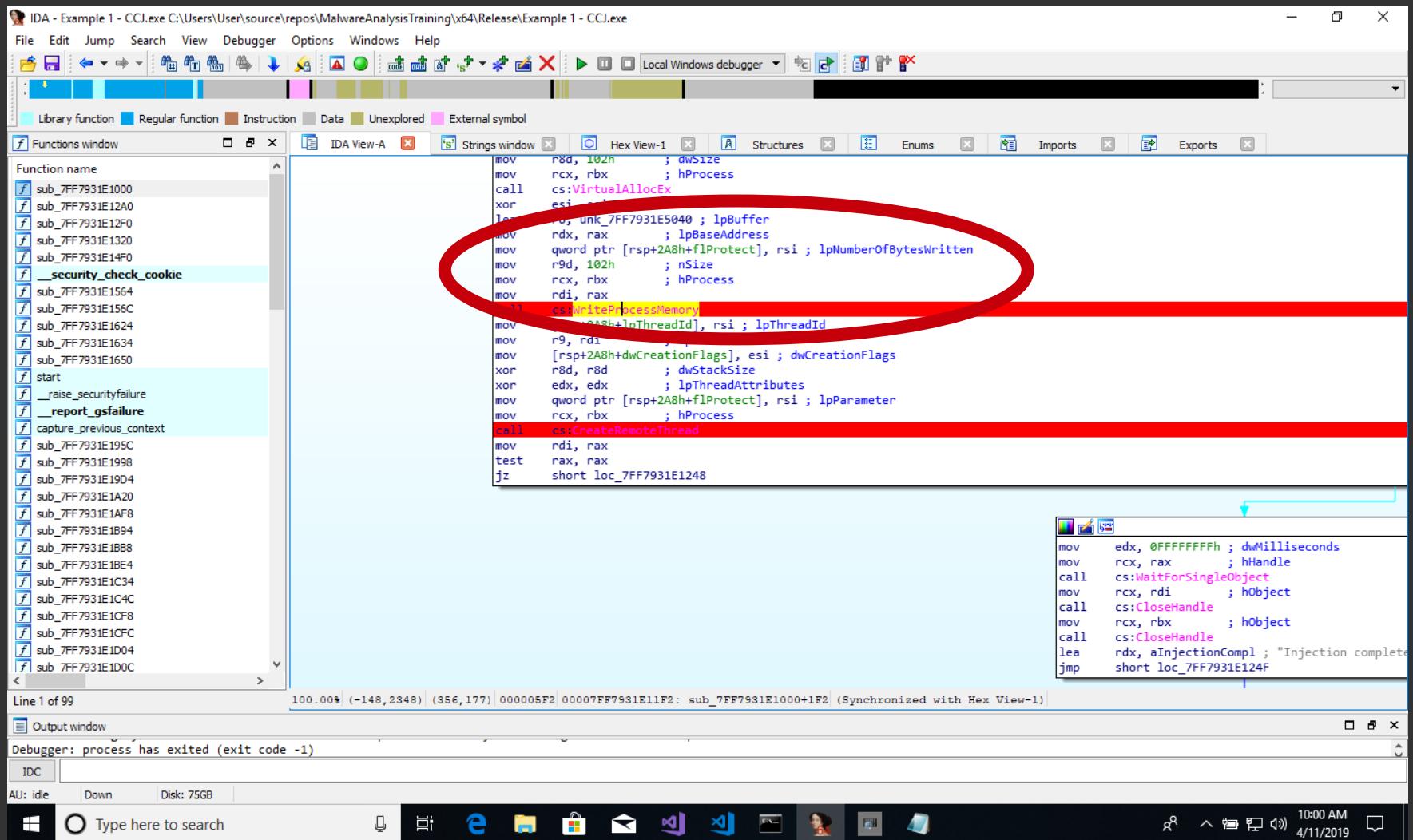
In the middle of the the “Home” view tab there's a restart button for notepad.

Click it to restart notepad, and then hit “Go” to restart the notepad from the start again.



Let's go into IDA and start the malware again, this time however we want to wait at the WriteProcessMemory breakpoint.

The reason, WriteProcessMemory has the location of what is being written into another processes memory. This is held in the “lpBaseAddress” value.



IDA - Example 1 - CCJ.exe C:\Users\User\source\repos\MalwareAnalysisTraining\x64\Release\Example 1 - CCJ.exe

File Edit Jump Search View Debugger Options Windows Help

Library function Regular function Instruction Data Unexplored External symbol

Functions window IDA View-A Strings window Hex View-1 Structures Enums Imports Exports

Function name

- sub_7FF7931E1000
- sub_7FF7931E12A0
- sub_7FF7931E12F0
- sub_7FF7931E1320
- sub_7FF7931E14F0
- security_check_cookie**
- sub_7FF7931E1564
- sub_7FF7931E156C
- sub_7FF7931E1624
- sub_7FF7931E1634
- sub_7FF7931E1650
- start
- __raise_securityfailure
- __report_gsfailure
- capture_previous_context
- sub_7FF7931E195C
- sub_7FF7931E1998
- sub_7FF7931E19D4
- sub_7FF7931E1A20
- sub_7FF7931E1AF8
- sub_7FF7931E1B94
- sub_7FF7931E1BB8
- sub_7FF7931E1BE4
- sub_7FF7931E1C34
- sub_7FF7931E1C4C
- sub_7FF7931E1CF8
- sub_7FF7931E1CFc
- sub_7FF7931E1D04
- sub_7FF7931E1D0C

100.00% (-148,2348) (356,177) 000005F2 000007FF7931E11F2: sub_7FF7931E1000+1F2 (Synchronized with Hex View-1)

Output window

Debugger: process has exited (exit code -1)

IDC

AU: idle Down Disk: 75GB

Type here to search

10:00 AM 4/11/2019

Running the process and continuing past the first breakpoint we end up at WriteProcessMemory.

The line with “lpBaseAddress”, highlighted in yellow, shows “mov rdx, rax” which translates to $rdx = rax$. Ignoring the assembly, it basically means rdx has the “lpBaseAddress”.

The screenshot shows the IDA Pro interface with the following details:

- Title Bar:** IDA - Example 1 - CCJ.exe C:\Users\User\source\repos\MalwareAnalysisTraining\x64\Release\Example 1 - CCJ.exe
- File Menu:** File Edit Jump Search View Debugger Options Windows Help
- Toolbar:** Local Windows debugger, various icons for file operations and analysis.
- Registers View:** Shows general registers (RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, RIP, R8, R9, R10, R11, R12, R13, R14, R15) with their current values.
- Stack View:** Shows the stack memory starting at address 00000000EC78FF520.
- Hex View:** Shows the raw hex dump of the assembly code.
- Output Window:** Displays log messages including "loaded C:\WINDOWS\SYSTEM32\VCRUNTIME140.dll", "thread has started (tid=1460)", and "PDBSRC: loading symbols for 'C:\Users\User\source\repos\MalwareAnalysisTraining\x64\Release\Example 1 - CCJ.exe'...".
- Bottom Bar:** Shows the status bar with "AU: idle", "Disk: 75GB", and the system tray with icons for volume, battery, and network.

The assembly code in the main window is:

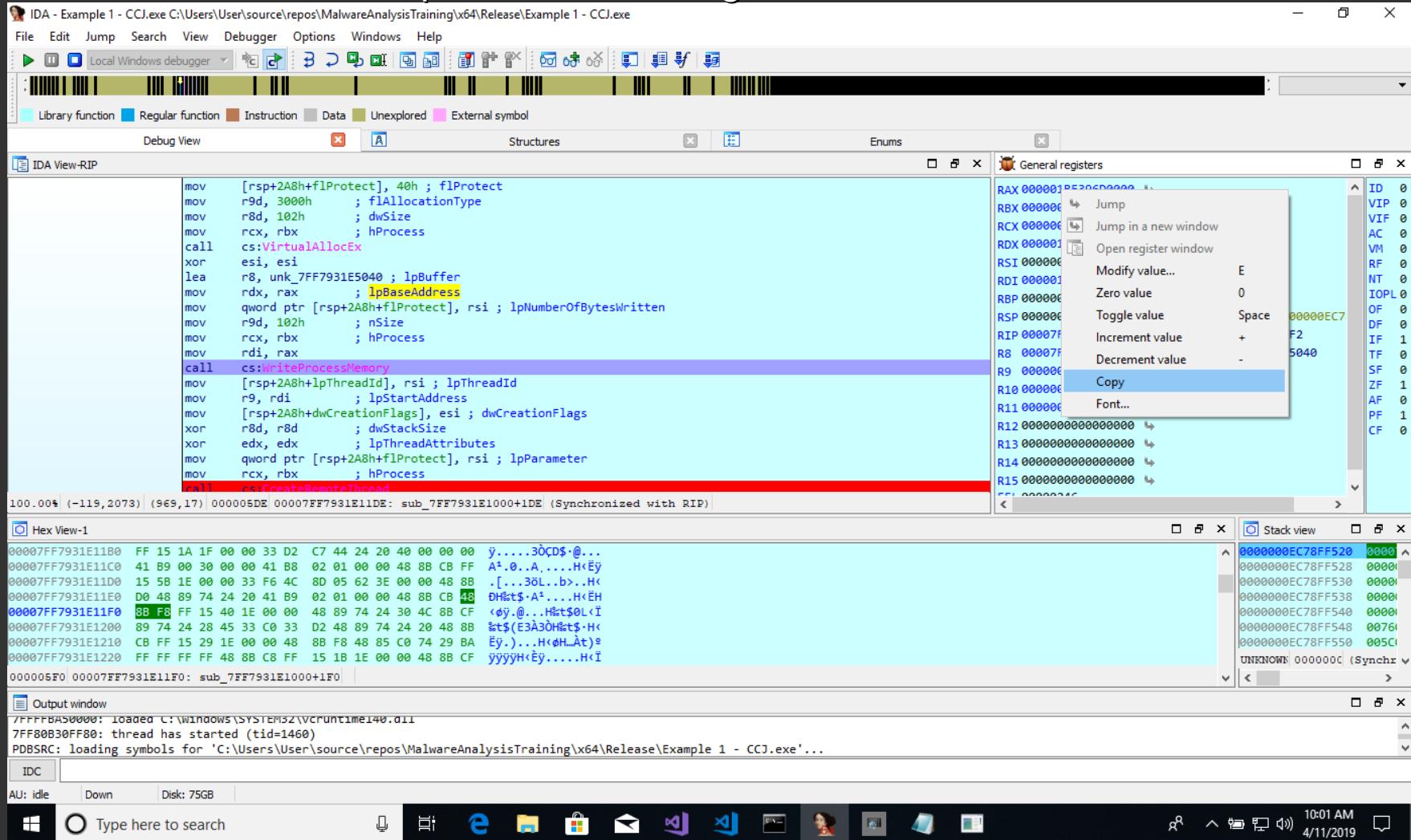
```
mov    [rsp+2A8h+f1Protect], 40h ; f1Protect
mov    r9d, 3000h ; flAllocationType
mov    r8d, 102h ; dwSize
mov    rcx, rbx ; hProcess
call   cs:VirtualAllocEx
xor    esi, esi
lea    r8, unk_7FF7931E5040 ; lpBuffer
mov    rdx, rax ; lpBaseAddress
mov    qword ptr [rsp+2A8h+f1Protect], rsi ; lpNumberOfBytesWritten
mov    r9d, [rdx=000001BF396D0000]
mov    rcx, rbx ; hProcess
mov    rdi, rax
call   cs:WriteProcessMemory
mov    [rsp+2A8h+lpThreadId], rsi ; lpThreadId
mov    r9, rdi ; lpStartAddress
mov    [rsp+2A8h+dwCreationFlags], esi ; dwCreationFlags
xor    r8d, r8d ; dwStackSize
xor    edx, edx ; lpThreadAttributes
mov    qword ptr [rsp+2A8h+f1Protect], rsi ; lpParameter
mov    rcx, rbx ; hProcess
call   cs:WriteProcessMemory
```

The line `mov rdx, rax` (which corresponds to `lpBaseAddress`) is highlighted in yellow.

The window on the right called “General registers” has the listing of all the registers and what is stored in them currently while debugging.

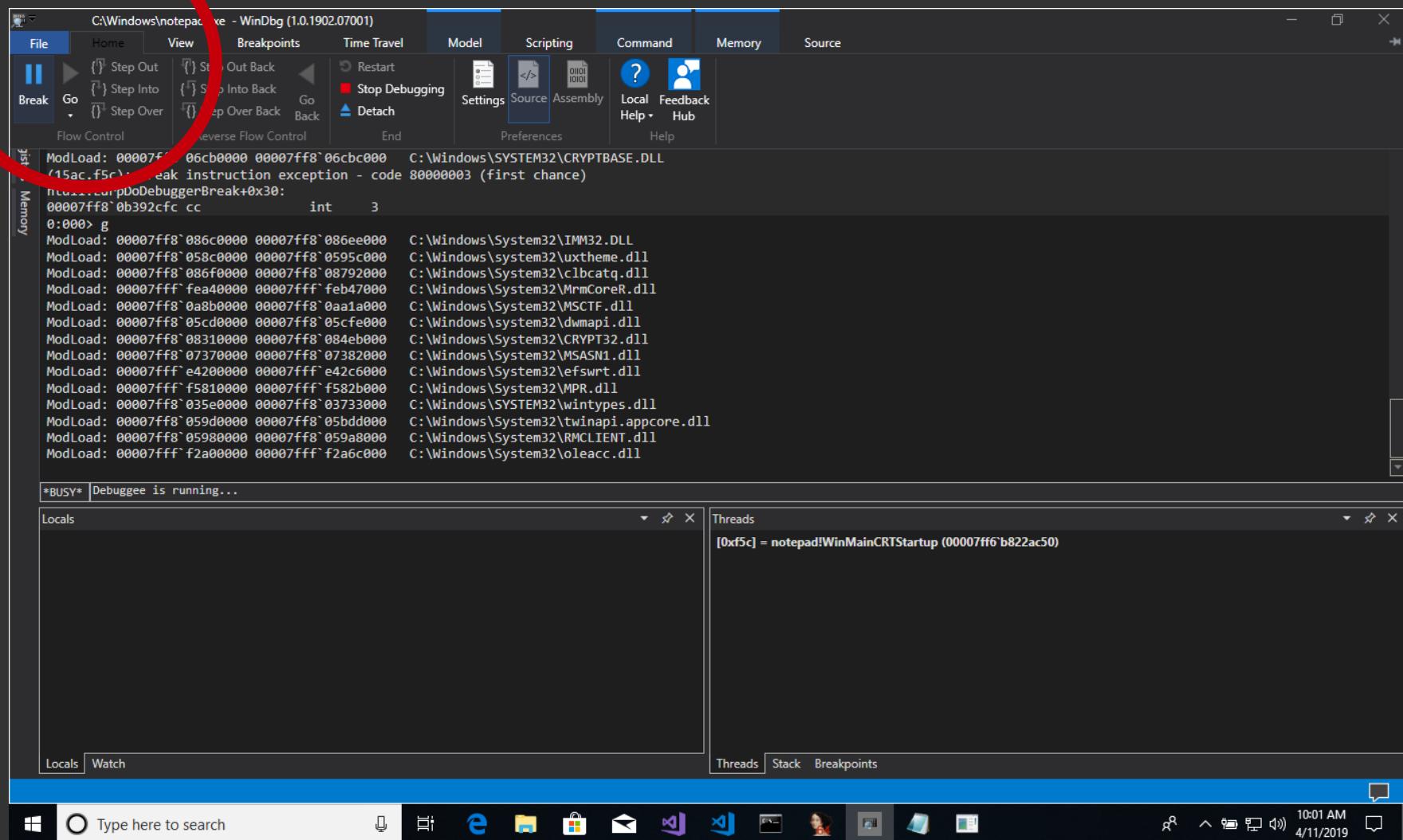
Let's hover over “RDX” and right click and click “Copy”.

We'll need this value to pull over to windbg.



Back in windbg we see a “Break” button next to the “Go” button.

Click the “Break” button to stop the notepad.



We're going to create a breakpoint in windbg now.

Typed “bp” in the console window and paste the value of “RDX” captured from IDA with a space in between them into the console.

```
C:\Windows\notepad.exe - WinDbg (1.0.1902.07001)
File Home View Breakpoints Time Travel Model Scripting Command Memory Source
Command X
Disassembly Registers Memory
0:000> g
ModLoad: 00007fff`fd3d0000 00007fff`fd678000 C:\Windows\SYSTEM32\iertutil.dll
ModLoad: 00007ff8`06cbc000 00007ff8`06cbc000 C:\Windows\SYSTEM32\CRYPTBASE.DLL
(15ac.f5c): Break instruction exception - code 80000003 (first chance)
ntdll!LdrpDoDebuggerBreak+0x30:
00007ff8`0b392cfcc cc int 3
0:000> g
ModLoad: 00007ff8`086c0000 00007ff8`086ee000 C:\Windows\System32\IMM32.DLL
ModLoad: 00007ff8`058c0000 00007ff8`0595c000 C:\Windows\system32\uxtheme.dll
ModLoad: 00007ff8`086f0000 00007ff8`08792000 C:\Windows\System32\clbcatq.dll
ModLoad: 00007fff`fea40000 00007fff`feb47000 C:\Windows\System32\MrmCoreR.dll
ModLoad: 00007ff8`0a8b0000 00007ff8`0aa1a000 C:\Windows\System32\MSCTF.dll
ModLoad: 00007ff8`05cd0000 00007ff8`05cfe000 C:\Windows\system32\dwmapi.dll
ModLoad: 00007ff8`08310000 00007ff8`084eb000 C:\Windows\System32\CRYPT32.dll
ModLoad: 00007ff8`07370000 00007ff8`07382000 C:\Windows\System32\MSASN1.dll
ModLoad: 00007fff`e4200000 00007ff8`e42c6000 C:\Windows\System32\efswrt.dll
ModLoad: 00007fff`f5810000 00007ff8`f582b000 C:\Windows\System32\MPR.dll
ModLoad: 00007ff8`035e0000 00007ff8`03733000 C:\Windows\SYSTEM32\winatypes.dll
ModLoad: 00007ff8`059d0000 00007ff8`05bdd000 C:\Windows\System32\twinapi.appcore.dll
ModLoad: 00007ff8`05980000 00007ff8`059a8000 C:\Windows\System32\RMCLIENT.dll
ModLoad: 00007ff8`059a8000 00007ff8`f2a6c000 C:\Windows\System32\oleacc.dll
(15ac.f418): Break instruction exception - code 80000003 (first chance)
ntdll!DbgBreakPoint:
00007ff8`0b363080 cc int 3

0:001> bp 000001BF396D0000
```

The parameter is incorrect. (0x80070057)

Locals Threads

[0xf5c] = notepad!WinMainCRTStartup (00007ff6`b822ac50)
[0x1418] = ntdll!DbgUiRemoteBreakin (00007ff8`0b38ef10)
[0x848] = ntdll!TppWorkerThread (00007ff8`0b30ff80)
[0xa50] = combase!CrpcThreadCache::RpcWorkerThreadEntry (00007ff8`0a61ef70)
[0x1684] = ntdll!TppWorkerThread (00007ff8`0b30ff80)

Locals Watch Threads Stack Breakpoints

10:01 AM 4/11/2019

Now hit “Go” again after making the breakpoint. Go back to IDA and continue running the program, and when you come back to windbg again you should see the breakpoint has been hit.

The screenshot shows the WinDbg interface for the process C:\Windows\notepad.exe. The Command window displays assembly code and memory dump information. A red circle highlights the line where a breakpoint was hit:

```
0:001> bp 000001bf396d0000
Breakpoint 0 hit
000001bf`396d0000 000605a       add    byte ptr [rax+5Ah],ah ds:000001bf`396d005a=1e
```

The status bar at the bottom indicates "The scope specified was not found. (0x8007013e)".

Below the Command window, there are tabs for Locals and Watch. To the right, there are tabs for Threads, Stack, and Breakpoints. The Threads tab shows the following thread list:

- [0xf5c] = notepad!WinMainCRTStartup (00007ff6`b822ac50)
- [0x848] = ntdll!TppWorkerThread (00007ff8`0b30ff80)
- [0x1684] = ntdll!TppWorkerThread (00007ff8`0b30ff80)
- [0x1390] = 000001bf396d0000

The system tray at the bottom right shows the date and time as 10:01 AM, 4/11/2019.

Let's type into the console terminal now in windbg "u". "u" stands for unassemble in windbg and allows us to see what code is where we are stuck at.

Looks like valid assembly, which means the WriteProcessMemory function is writing code into the notepad.

The screenshot shows the WinDbg interface with the following details:

- Command Window:** Shows assembly code starting with `00007ff8`0b363080 cc int 3`. The assembly listing continues with various instructions including add, push, pop, sub, mov, and lods.
- Registers:** Registers pane is visible on the left.
- Memory:** Memory pane is visible on the right.
- Locals:** Locals window shows the message: "The scope specified was not found. (0x8007013e)"
- Threads:** Threads window lists several threads:
 - [0xf5c] = notepad!WinMainCRTStartup (00007ff6`b822ac50)
 - [0x848] = ntdll!TppWorkerThread (00007ff8`0b30ff80)
 - [0x1684] = ntdll!TppWorkerThread (00007ff8`0b30ff80)
 - [0x1390] = 000001bf`396d0000
- Bottom Taskbar:** Shows the Windows Start button, a search bar with "Type here to search", and pinned icons for File Explorer, Edge, File History, Mail, Task View, File Explorer, and Task View again. The system tray shows the date and time as 10:02 AM on 4/11/2019.

Now let's take a step back and do some static analysis real fast.

Let's extract anything that looks like strings being held as numbers.

I see 636C6163h

The screenshot shows the WinDbg interface with the command window displaying assembly code. The assembly code includes several instructions involving the value `636C6163h`. The Registers and Memory panes are visible on the left, and the Threads pane shows four threads. The Locals and Watch panes are at the bottom. The status bar at the bottom right shows the date and time: 10:02 AM 4/11/2019.

```
C:\Windows\notepad.exe - WinDbg (1.0.1902.07001)
File Home View Breakpoints Time Travel Model Scripting Command Memory Source
Command X
0:0007ff8`0b363080 cc int 3
0:001> bp 000001BF396D0000
0:001> g
Breakpoint 0 hit
000001bf`396d0000 00605a add byte ptr [rax+5Ah],ah ds:000001bf`396d005a=1e
0:001> u
000001bf`396d0000 00605a add byte ptr [rax+5Ah],ah
000001bf`396d0003 6863616c63 push 636C6163h
000001bf`396d0008 54 push rsp
000001bf`396d0009 59 pop rcx
000001bf`396d000a 4829d4 sub rsp,rdx
000001bf`396d000d 65488b32 mov rsi,qword ptr gs:[rdx]
000001bf`396d0011 488b7618 mov rsi,qword ptr [rsi+18h]
000001bf`396d0015 488b7610 mov rsi,qword ptr [rsi+10h]
0:001> u
000001bf`396d0019 48ad lodsd qword ptr [rsi]
000001bf`396d001b 488b30 mov rsi,qword ptr [rax]
000001bf`396d001e 488b7e30 mov rdi,qword ptr [rsi+30h]
000001bf`396d0022 03573c add edx,dword ptr [rdi+3Ch]
000001bf`396d0025 8b5c1728 mov ebx,dword ptr [rdi+rdx+28h]
000001bf`396d0029 8b741f20 mov esi,dword ptr [rdi+rbx+20h]
000001bf`396d002d 4801fe add rsi,rdi
000001bf`396d0030 8b541f24 mov edx,dword ptr [rdi+rbx+24h]

0:001>
Locals Threads
The scope specified was not found. (0x8007013e)
```

I also see later after typing “u” in again. 456E6957h

Using a hex decoder we see the strings “clac” and “EinW”.

From experience this tells me this is code for a “WinExec(“calc”)” payload. So now we’ve generalized the payload and its behavior

The screenshot shows the WinDbg interface with the following details:

- File**, **Home**, **View**, **Breakpoints**, **Time Travel**, **Model**, **Scripting**, **Command**, **Memory**, **Source** tabs.
- Registers**, **Dereference**, **Registers**, **Memory** dropdowns.
- Command** pane: Shows assembly code for the notepad process. The code includes instructions like pop rcx, sub rsp,rdx, mov rsi,qword ptr gs:[rdx], and various mov and add instructions involving rsi, rdi, rax, rdx, rbp, and rbx registers.
- Locals** pane: Shows the locals stack frame. It contains the following entries:
 - 0fb72c17 movzx ebp,word ptr [rdi+rdx]
 - 8d5202 lea edx,[rdx+2]
 - ad lods dword ptr [rsi]
 - 813c0757696e45 cmp dword ptr [rdi+rax],**456E6957h**
 - 75ef jne 000001bf`396d0034
 - 8b741f1c mov esi,dword ptr [rdi+rbx+1Ch]
 - 4801fe add rsi,rdi
 - 8b34ae mov esi,dword ptr [rsi+rpb*4]
- Threads** pane: Lists the current threads:
 - [0xf5c] = notepad!WinMainCRTStartup (00007fff`b822ac50)
 - [0x848] = ntdll!TppWorkerThread (00007fff`80b30ff80)
 - [0x1684] = ntdll!TppWorkerThread (00007fff`80b30ff80)
 - [0x1390] = 000001bf396d0000
- Locals** and **Watch** tabs at the bottom of the Locals pane.
- Threads**, **Stack**, **Breakpoints** tabs at the bottom of the Threads pane.
- Taskbar icons: Search, Start, Edge, File Explorer, Mail, Snipping Tool, Task View, File History, Task Scheduler, Taskbar settings.
- System tray: Volume, Network, Battery, Date/Time (10:03 AM, 4/11/2019).

Example 1 - End

- At this point we've generalized the malware/binary enough from a start to end perspective.
- Any further analysis and we'd be wasting time on specifics.
- Your goal in every analysis step should be to turn the 100% of surface area to analyze to only 10% in each step.

